May 12, 2022 – Hands-On Simulating Stochastic Gene Expression
Prof. Joshua Weitz and QBioS Cohort

1 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, including Python. 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 zero 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 either in a row, or a column. 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 = 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 doesn't work, that's 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 `np.random.rand`? Similarly, what is the variance? Once you have identified the mean and variance, plot the distribution of numbers generated by `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?

Solutions to Challenge Problem: 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 $\mathrm{Var} = \langle x^2 \rangle - \langle x \rangle^2$, 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:

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

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

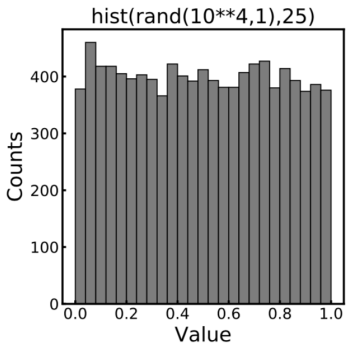
$$= 1/3 - (1/2)^2 \tag{3}$$

$$= 1/12 \tag{4}$$

This result can be verified entering the command `np.var(np.random.rand(10000))` which returns the variance of 10000 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 below 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 following 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.



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.

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. The following challenge problem provides an opportunity to build your intuition for manipulating and generating random numbers with distinct means and ranges.

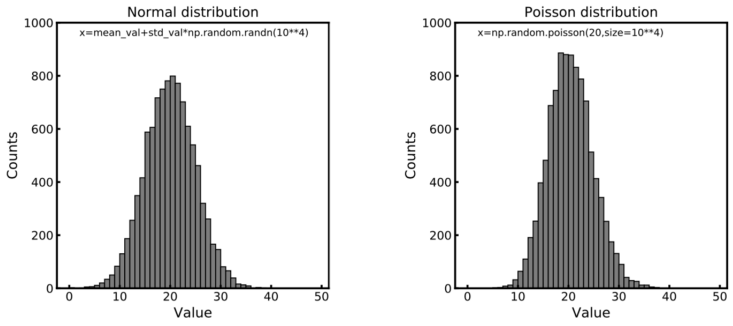


Figure 1: Sampling from random distributions, including the normal distribution (left) and the Poisson distribution (right).

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!

Solutions to Challenge Problem: 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 to 5...
  `np.random.rand(1000)*5`
- Generate 1000 random numbers equally spaced between 2 to 7...
  `np.random.rand(1000)*5 + 2`
- Generate 1000 random numbers equally spaced between -5 to 5...
  `np.random.rand(1000)*10 - 5`

2 Simulating Stochastic Gene Expression

Cellular gene regulation arises from the aggregative outcomes of many individual interactions, i.e., between a transcription factor and a promoter, an inducer and a protein, RNA polymerase and a genome sequence, etc. Each of these processes occurs again and again, so that the resulting dynamics are often represented as a type of average, or "mean-field" behavior. While there are computational methods to simulate deterministic dynamics of gene regulation, those simulation methods are applicable to other scales, whether of tissues, organisms, populations, or ecosystems, The accuracy of deterministic models improve as the number of

processes and number of molecules increases. But, one must proceed cautiously, particularly when the numbers are small, which is certainly the case when genes are turned 'On'.

Hence, we have a different goal in mind: simulating stochastic gene expression. Building upon the theory in the lectures, and the previous development of core programming skills, this document guide will help you understand how to: (i) translate gene expression processes into events; (ii) translate these events into a simulation; (iii) Simulate stochastic gene expression that recapitulates expected mean-field dynamics and that can be compared to measurements. The focal method of this lab is what is commonly known as the Gillespie algorithm. The Gillespie algorithm is a widely used approach developed for simulating chemical reaction kinetics, one event at a time. It can be used in many contexts, including stochastic gene expression. As will be explained below, the core idea of the Gillespie algorithm is that the state of the cell changes based on individual events. Each event is associated with a process, e.g., degradation or production. These processes have rates, such that it is possible to sample the exact time of the next event from an appropriate exponentially distributed distribution. Then, at the next event time, the state of the system changes. For example, if a production event occurs, then the number of proteins goes up by one. In contrast, if a degradation event occurs, then the number of proteins goes down by one. Given an updated system, the process repeats and repeats. A schematic of this process is depicted below with multiple realizations of the Gillespie algorithm on the right (see Figure 2).

Here then is the target. Consider a cell in which genes are transcribed and then the mRNA translated into protein. We will aggregate transcription and translation and assume that proteins are produced at a rate β, e.g., 50 or 100 nM/hr. The units are important. For example, a single protein in the volume of an *E. coli* cell is approximately 1 nM concentration. Because cells are growing, we will assume that proteins decay at an exponential rate of 1 hr^{-1}. This oversimplifies the actual change in concentrations due to cell growth, division, and active degradation but will suffice, for now. This lab will explore the consequences of tracking the dynamics of individual proteins rather than their concentrations. In doing so, the laboratory is centered around a series of questions. Will proteins decay away to nearly zero, increase without bound, or will they reach a steady state? In addition, given that stochasticity is now embedded into the core dynamics, we will need to revisit the very notion of a 'steady state'.

The lab focuses on the one-dimensional case, i.e., where the state space is represented by the numbers of proteins. However, the Gillespie algorithm is more general, and can also be used to simulate the stochastic

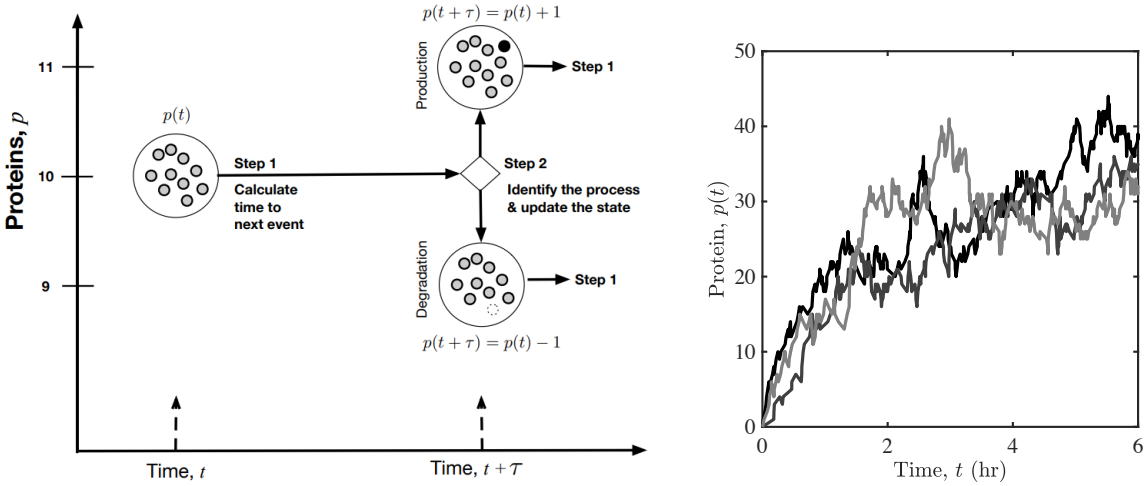


Figure 2: Gillespie algorithm from schematic to application. (Left) A schematic depicting the discrete change in the number of a state variable, in this case proteins. (Right) The result of implementing the Gillespie algorithm for the simplest model of gene expression.

dynamics of multi-component systems, e.g., those involving mRNA and proteins, or as part of feedback systems with mRNA, proteins, inducers, etc., and eventually as part of entire gene regulatory networks. In doing so, a secondary goal of this lab is to provide a gateway for simulating seemingly complex models that might not be so complex to simulate after all.

3 Poisson Processes: Finding the Time of the Next Event

3.1 Waiting time theory

How long does it take for the next production event to take place if the *rate* is b? This rate has units of inverse time, e.g., hrs^{-1}. There tends to be significant misunderstanding at what such units means. One way to think about them is that the inverse of the rate is a characteristic time, approximately equal to that of the time between random events. So events with a rate of 4 hrs^{-1} occur about every 15 minutes, whereas events with a rate of 10 hrs^{-1} occur about every 6 minutes. *Put simply: higher rates imply shorter intervals between events.*

Formally, this rate should be thought of as a *probability per unit time* that the event occurs. The probability that an event occurs in some small interval dt is $b \times \text{dt}$. The probability that an event does not occur is $1 - b \times \text{dt}$. For some, you might recognize such events as arising from a Poisson process. With rates defined, the question we want to ask is: *what is the probability that the event takes place at some time t from now.* For example, if an event occurs at a rate of 5 per hour, then what is the probability that the *next event* occurs precisely 0.2 hours from now, or at 0.4 hours, or 2 hours from now?

For an event to occur at time t for the first time, it should not occur anytime before-hand. This seems obvious, but it's the key to moving from rates to probabilities. Recall that $p(t)\text{dt}$ is the probability that the event occurs between time t and time $t + \text{dt}$ where dt is some small time interval, e.g., 0.00001 hrs. Therefore, for the event to occur at time t, then it should satisfy the following equation:

$$p(t)\text{dt} = \overbrace{(1 - b\text{dt}) \ldots (1 - \text{bdt})}^{\text{Did not occur } \frac{\text{t}}{\text{dt}} \text{ times}} \times \overbrace{b\text{dt}}^{\text{did occur}} \tag{5}$$

In words, this means that the event does not occur in any of the $\frac{t}{\text{dt}}$ intervals between 0 and t. Then, the event does occur with probability $b$dt in the time interval between t and $t + \text{dt}$.

Next, keep in mind that $b\text{dt} \ll 1$, i.e., the probability is very small given small time intervals. Hence, we can approximate $1 - b\text{dt} \approx \text{e}^{-\text{bdt}}$. Combining this together we find

$$\begin{aligned} p(t)\text{dt} &= e^{-b\text{dt}\frac{\text{t}}{\text{dt}}} \cdot b\text{dt} \\ p(t)\text{dt} &= e^{-bt} b\text{dt} \end{aligned}$$

This is the key result. A process that occurs at a rate b leads to *exponentially* distributed events. This is termed a *Poisson* process, i.e., one in which the average time between events is fixed, but the exact time of the next event is unknown and sampled from a memoryless (exponential) distribution.

3.2 Some computing

There are multiple ways to generate exponentially distributed random numbers. The easiest is to use the built-in random number generator in Python:

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
b=3
np.random.exponential(1/b)
```

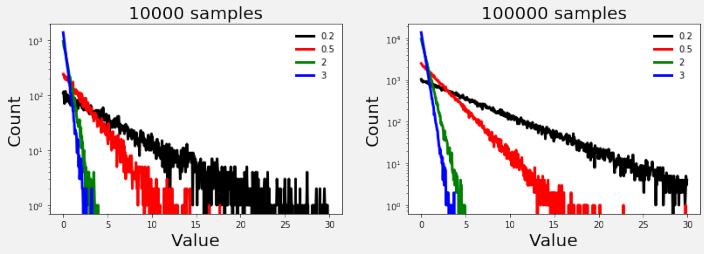
Here, the rate is 3 per hr, and the function `np.random.exponential` takes a single input – the inverse of the rate – and returns an exponentially distributed random number. Hence, the input argument to `np.random.exponential` should be the expected mean of the exponentially distributed set of random numbers. Verify that this works by generating 10,000 such random numbers, following the code snippet below to find their mean:

```
tvals = np.random.expoential(1/b,10000)
meant = np.mean(tvals)
```

As you notice, the mean you measure not *exactly* what you expected given the input rate parameter – 1/3? This is expected, given that these are chosen randomly, but the mean is indeed very close to 1/3 and will be ever closer the more samples you take (at least on average). Next up: a challenge problem.

Challenge Problem: Exponentially Distributed Random Numbers

Plot the probability distribution for 10^4 and 10^5 exponentially sampled random numbers. Are the realized distributes in fact exponential? Now, change the rate parameter, e.g., setting it to $b = 0.2$, 0.5, 2 or 3. Recall that the `hist` command is one way to visualize distributions, although there are other options as well. If your code works, the result should look something like the following:



Solutions to Challenge Problem: Exponentially Distributed Random Numbers The following code generates samples of 10^3, 10^4 and 10^5 random numbers (by varying the variable `num`). One could plot the cumulative distribution function (cdf) or the probability distribution function (pdf). This code uses the pdf and it is apparent by plotting the histograms on logarithmically scaled y-axes that the distributions are in fact exponential. The slopes of the straight lines correspond to the values of b, denoted in the legend. Note that the estimates of the rate parameter, correspond closely but not exactly to the generating value. However, the 95% CI-s are almost certainly going to include the mean – see the text for details.

```
# Setup
num = 10000
bvals = [0.2, 0.5, 2, 3]
b_est = np.zeros(np.shape(bvals))
b_ci = np.zeros((len(bvals),2))
tmpcol = ['k','r','g','b']

# Loop for each conditions of b
for i in range(len(bvals)):
    #generate plots
    b=bvals[i]
    #sample at random
    x=np.random.exponential(1/b,num)
    #take a histogram with uniform bins
    figTemp = plt.figure(1)
    bins = np.arange(0,30.05,0.05)
    [tmpcount,tmpx,patches]=plt.hist(x,bins)
    figReal = plt.figure(2)
    plt.semilogy(tmpx[:-1],tmpcount,color=tmpcol[i],linewidth=3)

# Plot labeling
plt.xlabel('Value',fontsize=20)
plt.ylabel('Count',fontsize=20)
plt.title('{num} samples'.format(num=num),fontsize=20)
plt.legend(list(map(str, bvals)),frameon=False)
```

4 A Theory of Timing Given Multiple, Stochastic Processes

The prior section delineates the relationship between the rate of a Poisson process and the timiing of the next event. But, what happens if more than one kind of event can take place independently, each with their own characteristic rate? What then is the expected time before any event of either kind takes place? This next section addresses the issue of identifying the time before *either* process 1 occurs OR process 2 occurs. For example, if proteins can be produced or degraded, then seemingly some event will happen faster than when only consider prodution alone or degradation alone. Note that consistent with the usage thus far, we will assume units of hrs for timing and hrs^{-1} for rates.

To begin, consider two independent Poisson processes with rates $b_1 = 0.5$ and $b_2 = 2.5$. Given those two rates, what is the expected time until the next event? This question can be answered computationally as a guide to the intuition developed in the main text. One way to answer this equation is to sample two exponentially distributed random numbers using `np.random.exponential` (or using the conversion formula leveraging `np.random.uniform` above). Then, the waiting time should correspond to the *smaller* of these two waiting times, as in:

```
# Set the rates
b1=0.5
b2=2.5
# Generating the anticipated event times
r1=np.random.exponential(1/b1)
r2=np.random.exponential(1/b2)
```

```
# Find the smaller time
etime1 = min(r1,r2)
```

The smaller time stored in `etime1` corresponds to the event that occurs first. The function `min` selects the minimum value amongst the two exponentially distributed random numbers. As an intuitive check, run the code and then ask yourself: was the number closer to 2 (the inverse of `b1`) or closer to 0.4 (the inverse of `b2`)? In reality, a single event is insufficient to help build up one's intuition. Instead, repeat the above procedure 1000 times, recording the times in an array. Here is one way to do it:

```
etime = np.zeros(1000)
for i in range(1000):
    b1=0.5
    b2=2.5
    r1=np.random.exponential(1/b1)
    r2=np.random.exponential(1/b2)
    etime[i] = min(r1,r2)
```

Irrespective of how you calculated it, consider the following two questions:

- What is the distribution of the variable `etime`, in other words the expected time to the first event?
- What is the average time to the first event?

Whether plotting the pdf using the `hist` command, or using the cdf (defined above), you will notice that the events appear exponentially distributed. They are! The reason is that rates of events add. In other words, the next event occurs as if there was a single Poisson process with rate $b = b_1 + b_2 = 3$ per hour! This means that the average time is closer to 1/3 of an hour (i.e., faster than either process alone!).

Formally, we can compare the distribution to the expected distribution from a Poisson process with $\lambda = 3$. When comparing continuous distributions it is often useful to compare the cumulative distributions. If the data of times between events is saved as a variable `etime`, then the following code snippet compares the realized cdf vs. the theoretically expected cumulative distribution function. Define $Q(t)$ as the cumulative distribution function, i.e., the probability that the event happens at any time between $0 \leq t' \leq t$, then

$$Q(t) = \int_0^t \mathrm{dt'p(t')} = \int_0^t \mathrm{dt'be^{-bt'}}, \tag{6}$$

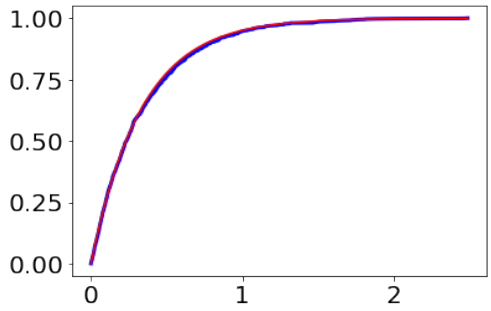
$$= -e^{-bt'}\Big|_0^t, \tag{7}$$

$$= 1 - e^{-bt}. \tag{8}$$

The empirical cdf can be calculated by sorting the n sampled values, and finding the fraction of the values whose timing is less than that, for every point $1 \ldots n$. The following code compares the empirical cdf to the theoretical expectation. Note that this code is does not include comments. Try to figure it out yourself, and then look at the documented version just afterwards.

```
fig=plt.figure()
ax=fig.gca()
sortdata = np.sort(etime)
cdfvals = np.arange(1,len(sortdata)+1)/len(sortdata)
plt.plot(sortdata,cdfvals,color='b',linewidth=3)
poiss3cdf = lambda t: 1-np.exp(-3*t)
plt.plot(sortdata,poiss3cdf(sortdata),'r')
plt.setp(ax.spines.values(),linewidth=3)
ax.tick_params(labelsize=20)
```

The result of this code should lead to the following:



This code has done a few things. Let's diagnose it, including comments for added explanation.

```
fig=plt.figure()
ax=fig.gca()
# This sorts the event times from smallest to largest
sortdata = np.sort(etime)

# This determines the fraction of the data less than a particular value
# This is the cumulative distribution
cdfvals = np.arange(1,len(sortdata)+1)/len(sortdata)

#This next line creates a function, with one variable t that returns
# the CDF for an exponential distribution with rate parameter 3
exp3cdf = lambda t: 1-np.exp(-3*t)

# This plots the CDF of the sorted times
plt.plot(sortdata,cdfvals,color='b',linewidth=3)

# This plots the CDF of the predicted exponential distribution
plt.plot(sortdata,exp3cdf(sortdata),'r')

# This is an example of how to change label axes sizes
ax.tick_params(labelsize=20)
```

With this code complete, it is time for the next challenge.

Challenge problem: Identity of Random Events

This challenge is centered on solving the following: given two independent Poisson processes operating concurrently, what fraction of times do each of the two processes take place first? That is to say, if $b_1 = 0.5$ and $b_2 = 2.5$, what is the fraction of events associated with process 1 and what fraction of events are were associated with process 2? Write your own code to try and solve this, leveraging the code already written thus far in the laboratory guide.

Solutions to Challenge Problem: Identity of Random Events. The theoretical expectation is that a Poisson event that occurs at a rate b_i amongst S events occurs before all others a fraction $\frac{b_i}{\sum_i b_i}$ of the time. Hence, in the case where $b_1 = 0.5$ and $b_2 = 2.5$, then the slowere process 1 will occur 0.5/3 or 1/6-th of the time and the faster process 2 will occur 2.5/3 or 5/6-th of the time. The following code snippet shows up to do this, even if you did not know or are not convinced by this argument. The lecture notes provide a detailed derivation of why this claim holds in general.

```
#Generate random numbers
num=10**3
b1=0.5
b2=2.5
b1_rnd=np.random.exponential(1/b1,num)
b2_rnd=np.random.exponential(1/b2,num)
#Find out which came first
tmpi1=np.where(b1_rnd<b2_rnd)[0]
tmpi2=np.where(b2_rnd<=b1_rnd)[0]
print('Fraction of 1-events that come first is',len(tmpi1)/num)
print('Fraction of 2-events that come first is',len(tmpi2)/num)
```

This section, and challenge problem, suggests an alternative method of simulating. Instead of choosing the shortest time, we could first independently sample the time according to a Poisson process with a rate corresponding to any process occurring. In general, the rate of any process occurring is $\lambda = \sum_j \lambda_j$, where λ_j is the rate of a single process, j and the sum includes all subprocesses. Next, we would sample which reaction occurs. The subprocess, j, is chosen with probability $\frac{\lambda_j}{\sum_j \lambda_j}$, where the sum includes all subprocesses. This approach is at the core of the Gillespie algorithm.