**Lab 6**

# Random processes – diffusion and gene expression

Quite a bit of what occurs inside a cell involves randomness, or *stochasticity*. In many cases, important processes rely on the *diffusion* of particles—their random motion throughout the cell. A few examples include the production of mRNAs and the feedback inhibition of a gene (when the protein encoded for by the gene finds the gene turns it off).

A common metaphor for diffusion (and, indeed, a regular name in the mathematical literature) is a *drunken walk*. Much like an embarrassingly drunk individual, a biomolecule has no control over whether it is about to go left or right. We can think about its movement one step at a time, and the particle is equally likely to go left or right with each.

# Simulations of a Random Walk

If a random walker takes ten steps, what do you think is its most likely final location? How about if it takes 100 steps?

A random walk will generate a probability distribution, with the most likely final location being the starting point. For 10 steps, the probability of ending at the starting point will be around 0.25; for 100 steps, the probability will be about 0.08.

In order to learn something about a random process, you have to take samples—and it’s much easier to generate lots of samples using a computer than to observe them with (more realistic) experiments. Let’s start with a single sample (one random walk), and let’s start with ten steps, in order to explore the question you just answered.

To keep your code organized and easy to execute, create a new script in Matlab. You can use this script to compile and edit your commands for the next few sections.

Begin by defining a variable *x* in MATLAB and setting it equal to zero—in other words, the particle’s starting position is zero. Now, create a “for” loop like you have in previous labs, which carries out the following steps ten times:

r =rand();

if r < 0.5

rest of the code

The “rest of the code” should execute an “if-else” statement (again, like Lab 1), which does the following: if *r <* 0*.*5, *x* is decreased by one, and if *r >* 0*.*5, *x* is increased by one. The block of code should end with two *end* statements. Paste your code below.

%%

x = 0;

for i = 1:10

r =rand();

if r<0.5

x= x-1;

else x=x+1;

end;

end

x

Run this block of code first for ten steps and then for 100 steps. What is the final location of the particle in both cases? Are those reasonable results?

For 10 steps, final value was -2; for 100 steps, it was -6

# Generating a Large Number of Samples

In the last section, you simulated two random walks with different numbers of steps. Now, your goal is to learn something about this random process using statistics. The statistical tools at your disposal include the mean, the standard deviation, and a histogram of the collection of samples. It’s important to remember that statistics tell us nothing definitive (“deterministic”) about a process; instead, they allow us to make predictions about an outcome based on experience—under the *assumption* that the system will behave in the same way in the future as it has in the past.

The first thing that you have to do is to modify the code that you’ve already written to make it simulate multiple random walks at once. Define a variable called *number\_of\_walks* and set it equal to 1000. You’ll also need to *initialize* a vector to keep track of the locations of all 1000 particles—use

>> positions = zeros(1, number\_of\_walks);

Here are the changes that you need to make to the code that generates a single random walk:

* Add a second “for” loop around (before) the current one, which iterates from *j* = 1 to *number\_of\_walks*
* Reset *x* to zero at the beginning of each walk
* Store the result of each random walk in the appropriate element of *positions*
* That’s it!

Make sure that you have the proper number of *end* statements.

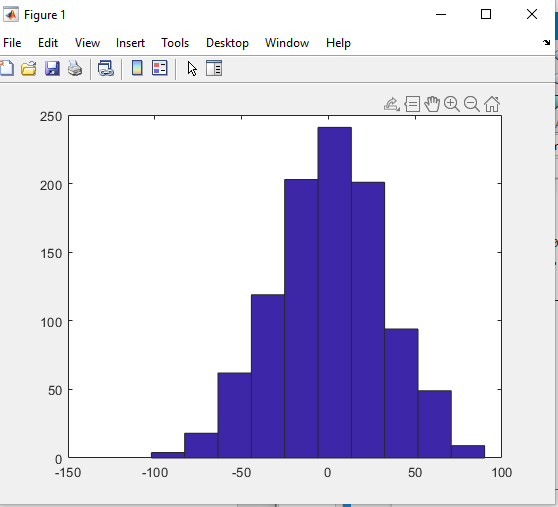
# What Can We Learn from the Data?

A *histogram* is a very useful tool for visualizing the results of sampling in statistics. A simple histogram divides all the possible values of the random variable into, say, ten “bins” and then counts how many samples fell into each bin. For example, if your simulations produce fifteen random walks ending at locations between *x* = 4 and *x* = 6, then your histogram will be a *bar plot*, whose bar from *x* = 4 to *x* = 6 is fifteen units high. Histograms are a very intuitive way to present data.

MATLAB can produce a simple histogram for samples contained in a vector by automatically choosing ten bins of equal size. Produce this plot for the data that you just created:

>> hist(positions);

Paste a screenshot of your histogram below:



MATLAB also allows you to specify the bins you want precisely. You can do that with the following line:

>> hist(positions,-50:50);

Based on this histogram, what is the most likely ending location for a particle?

36

What else about this plot makes sense?

The most likely ending location looks like it’s at 6, pretty close to the original position.

What do you predict will happen if you increase the number of walks? Justify this prediction.

The most likely location should get closer and closer to 0, as the number of trials increases. The width of the histogram will get wider but the values will still cluster around 0.

Try simulating ten or 100 times as many walks, and produce a histogram for this data. Is your plot symmetric? Do you expect it to be?

It’s broadly symmetric at 10000 walks, with two peaks at -28 and 2. The shape is roughly the same is for 10,000 walks as for 1,000 walks.

Histograms “coarse-grain” the data somewhat—this means that you group together data points that are similar in order to simplify your analysis. Of course, the simplest possible representation of your data would be a single number, and the *mean* and the *standard deviation* are two such options, which are sometimes useful for summarizing your results.

Calculate the mean and the standard deviation for one set of random walks using:

>> mean(positions);

>> std(positions);

List your results below.

For 10,000 walks, the mean was -2.2482 and SD was 99.5670

Compare these two numbers to those for another set of random walks with either a larger or a smaller number of particles. How do the means and standard deviations compare in both cases? Can you provide an explanation for this?

For 1,000 walks, mean was -0.352 and SD was 31.4357. The values still cluster around 0, but individual values are still random.

# Random numbers with other distributions

The function *rand()* generates random numbers distributed evenly between zero and 1. However, we often want to generate random numbers that come from other distributions. In particular, in class we talked about the binomial distribution and the Poisson distribution, both of which are very commonly found in biology. MATLAB has built-in functions that will allow you to generate these random numbers as well.

The function *poissrnd* generates random numbers distributed according to the Poisson distribution. It takes one or more inputs: the expected or average value *lambda* that characterizes the Poisson distribution, and optional inputs to specify the size of a vector or matrix of random numbers to be returned. Try the following:

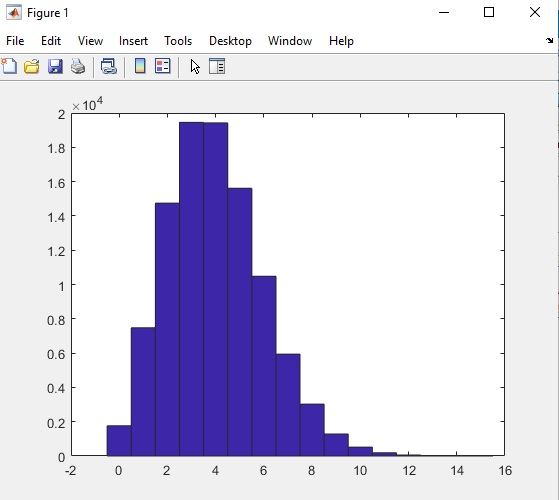
>> rand([1 10])

>> poissrnd(3,[1 10])

What do you notice about the random numbers returned by *poissrnd*? How are they different from those returned by *rand*?

Rnd returns floating-point numbers, while poissrnd returns integers.

Now let’s make a big list of random numbers using the function. Create a variable *vals* as a 100,000 x 1 array of random numbers from a Poisson distribution with a *lambda* value of 4. Generate a histogram of *vals* with bins of 0:15. Paste in a screenshot of your histogram.



Compute the mean and standard deviation of your list of randomly generated numbers. For the Poisson distribution, the mean should be lambda, and the standard deviation should be sqrt(lambda). Was that (approximately) the case?

Yes, mean was 3.997 and SD was 2.003.

The *binornd* function does essentially the same thing for the binomial distribution. However, the binomial distribution requires different input parameters to specify the distribution. It requires the total number of trials and the probability of a “success” on each individual trial. We will repeat the above exercise using the binomial distribution. Write one line of code to make vals a 100,000 x 1 vector with random binomially distributed values for *n=10* trials and *p=0.4* (probability of success). Paste your code below:

vals=binornd(10,0.4,[1 100000])

You can verify that your code gives you a list of 100,000 numbers. As before, generate a histogram of the generated numbers. In this case, what is the largest value in the list? (You can use the *max* function to determine the answer). Why was that the largest value?

Maximum value is 10, because n in the binomial distribution is 10.

What were the mean and standard deviation of your list? How do these values compare to those you got using the Poisson distribution? Why were they different or the same?

Mean was 3.9969 and SD was 1.5523, quite similar to the Poisson numbers. This isn’t surprising since Poisson distributions are basically binomial distributions with large n and small p.

# Building a Stochastic Model

We are now ready to start using randomly generated numbers to develop a stochastic model of a biological process. We will begin by making a stochastic version of the simple transcription model we built last week. Download and open the provided m-file “Stochastic\_Transcription\_1.m”.

Look over the code. As we did last week, we define rate constants for transcription, translation, and degradation of each molecule. However, this time instead of using the rates in differential equations, we will interpret the rates as probabilities or expected number of events per unit time.

Under the heading “Simulate”, two lambda values are defined. Looking at the code, what is the meaning or interpretation of *lambdaM*? Why is it defined to have that value?

LambdaM is the number of transcripts made in a particular time interval. It’s the product of the transcription rate and the time period (rate is transcripts/min or 30 s or whatever, and the time interval is the number of minutes or intervals, etc).

In the next two lines, we define probabilities for degradation of mRNA and protein molecules in each time step in a similar way. Then we update the number of mRNA molecules and protein molecules at each time step. Why do we use the Poisson distribution for the positive term and the binomial distribution for the negative term? (What could go wrong if we used the Poisson distribution for degradation?)

If Poisson was used for degradation, we might end up modeling the degradation of more mRNA than has been produced.

Run the blocks to simulate the model and display the results. How do the graphs for mRNA and protein compare?

mRNA is a lot noisier than the protein.

In our continuous model, what was the equation for the steady-state mRNA level?

m = r/g

On average, does the stochastic model reach the same typical value?

Yes

Where the stochastic model really differs from the continuous model is in our estimation of the “noise.” Whereas the continuous model perfectly approaches steady state, the stochastic model always has fluctuations. There are two common measures of how “big” this noise is: the standard deviation and the coefficient of variation or “CV”. The CV is defined as the standard deviation divided by the mean.

Edit the second block to uncomment the last five lines and make it compute and display the mean, standard deviation, and CV for mRNA and protein levels. You can exclude the first ~100 time points when the system is still approaching its steady-state-like behavior. Paste your code below.

(The fprintf command prints “formatted text”. The “%.1f” and “%.2f” are codes to say that a “floating point” or decimal number should be inserted with either 1 or 2 decimal places. The “\n” means to start a new line. The extra inputs at the end will be the values to insert where codes like “%.1f” were.)

fprintf('\nmRNA statistics\n');

fprintf(' mean = %.1f Std Dev = %.1f CV = %.2f\n', mean(m), std(m), std(m)/mean(m));

fprintf('\nProtein statistics\n');

fprintf(' mean = %.1f Std Dev = %.1f CV = %.2f\n', mean(p), std(p), std(p)/mean(p));

Rerun the model with different values for the transcription rate. How do the standard deviations and CVs change?

SD is slightly smaller than the square root of the mean

With transcription rate = 5: mean mRNA = 19.9; SD = 4.9; CV = 0.25

10: 40.4; 7.1; 0.18

1: 3.9; 1.8; 0.47

For a “Poisson process”, we expect the standard deviation to be equal to the square root of the mean. In many cases, this is something like an intrinsic “floor” or minimum in terms of the noise possible for a system. Processes for which the standard deviation is greater than the square root of the mean can be considered “noisier” than a Poisson process. Does the mRNA level fit the expectation for a Poisson process? (It may help to compute the standard deviation divided by the square root of the mean, and repeat the simulation a few times to see how this value varies). Why do you think this is or is not the case?

Seems close to a Poisson process: SD/sqrt(mean(m) = 1 when m = 1

1 when m = 5

1.1 when m = 10

N is large and p is small.

Does the protein level fit the expectation for a Poisson process? Why do you think this is or is not the case?

Seems pretty far from Poisson: SD/sqrt(mean(p) = 2.6 when p = 1

= 6.5 when p = 5

= 7.8 when p = 10