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

1 Gillespie algorithm applied to a gene expression model

1.1 Conceptual framework

The previous section demonstrated how to identify the waiting times between events given two Poisson processes with constant rates. But rates can also depend on the state of the system. For example, in a gene regulatory system, many reactions occur at rates proportional to the concentration of molecular types, e.g.,:

- Substrate and enzyme
- Transcription factor and DNA binding site
- Receptor and ligand

There is another difference between the next challenge and the prior work. When an event occurs inside the cell, then the degradation, production, or transformation of a molecule changes the state of the system and therefore has the potential to alter the underlying rates of the governing processes. This feedback is at the core of the Gillespie algorithm.

Returning to the motivating example, consider a cell in which proteins are produced at a constant rate β and decay at a rate α. The abundance of proteins, p, can be represented in terms of a differential equation:

$$\frac{dp}{dt} = \overbrace{\beta}^{\text{production}} - \overbrace{\alpha p}^{\text{decay}}. \tag{1}$$

In general, nonlinear equations of gene regulatory dynamics can't necessarily be solved analytically, though this one can be. However, it is possible to simulate dynamics arising from such equations via numerical integration. If one numerically integrates this equation given $\beta = 30$ nM/hr^{-1} and $\alpha = 1$hr^{-1}, the system converges to a fixed amount. Even without integrating, it should be apparent that there is a protein abundance at which production balances decay. That balance happens when $p^* = \beta/\alpha$.

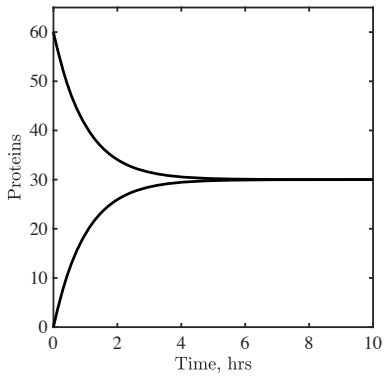


Figure 1: Dynamics of protein concentration given continuous model, given two initial concentrations, 0 and 60 proteins, and an equilibrium of 30.

But, is this what happens in a real cell where there can be 0, 1, 2, ..., 29, 30, 31, ... of proteins but not 29.231312 proteins? How can a stochastic algorithm capture the random nature of protein production? According to the Gillespie algorithm, the total rate at which any event occurs is the sum of the rates of individual processes:

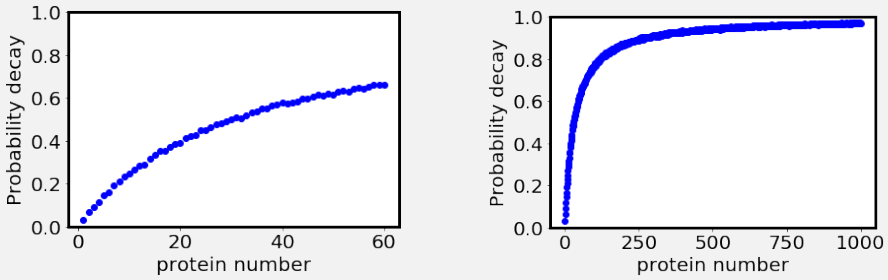
Production Occurs at a constant rate β

Decay Occurs at a protein-dependent rate αp

Based on this, first try to sketch the probability the next event is a decay event as a function of the number of proteins, does it go up, down, or remain constant as a function of p.

Challenge Problem: State-Dependent Rates

How does the probability of decay change as a function of the state p? In addition, at equilibrium, $p^* = \frac{\beta}{\alpha} = \frac{30}{1}$, what is the probabiility of decay at equilibrium? Use code and visualizations to support your argument. If your code is working, it should look like this, depending on whether you visualize from 0 to 60 proteins (left) or from 0 to 300 (right):

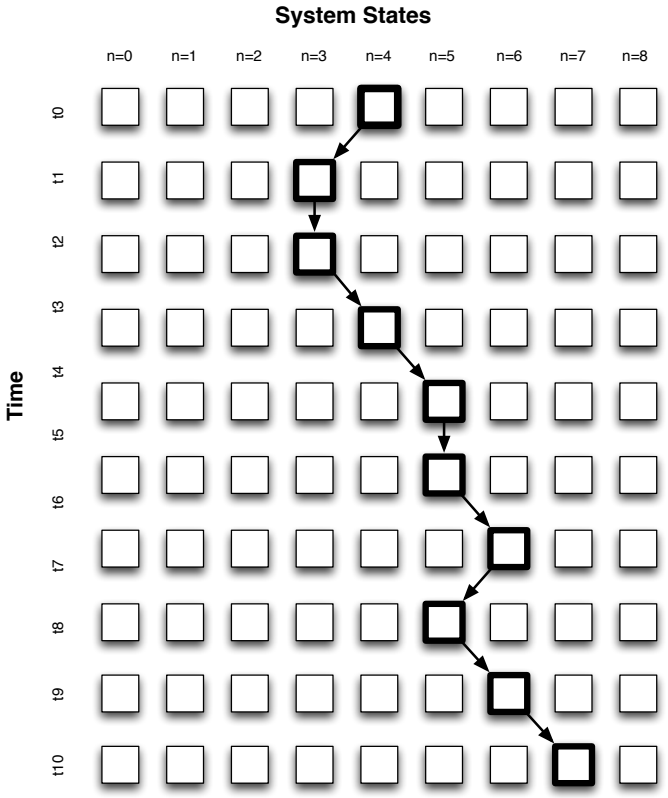


1.2 Gillespie algorithm

Before implementing the full algorithm, it is worth discussing the algorithm in a generic form. Computer scientists often call this "pseudocode". Pseudocode is not written in a specific language but nonetheless lays out a series of steps that your actual code might follow. Here is a pseudocode version of the Gillespie algorithm:

```
specify time range of simulation
set initial time vector and protein numbers vector to 0
while the current time is less than the maximum
        update rate of decay based on number of proteins
        update rate of production
        find the waiting time until the next event
        update the current time based on the waiting time
        decide on which process occurred
        update and store the number of proteins based on selected event
        store the information about the system state at the current time
```

This pseudocode represents an ideal time to pause, strategize, and ideally discuss each step with a classmate or study partner. Try to draw a sketch of what might happen given a cell that has initially 4 proteins vs. one in which there are initially 1 protein. Is every trajectory the same? Note that in this case, the rate of production remains constant irrespective of the current protein level. This need not be the case generally, e.g., as part of gene regulatory networks in which the current expression levels influence new production. In doing so, the following schematic may be helpful. In reflecting on its meaning, also consider: what would happen if the system started with 0 proteins, could the gene still turn 'On'?



1.3 Implementing stochastic gene expression

It is time to build a model of stochastic gene expression using the Gillespie algorithm. The following is one version, feel free to follow it or implement a version on your own! Start a script, and enter the following (though the comments are optional):

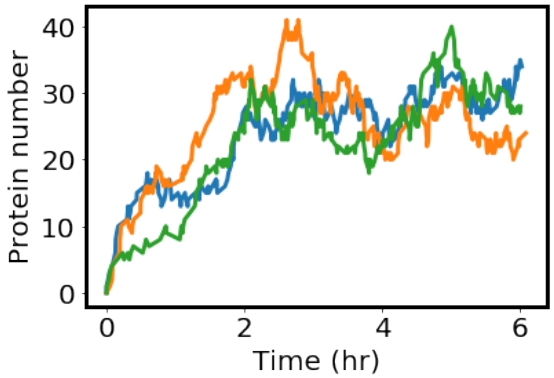
```
#stochastic gene expression
#part 1 - initial conditions
p0=0 #initial proteins
alpha = 1 #decay rate
beta = 30 #production rate
currp = p0 #current proteins
tmax = 6 #max time
currt = 0 #current time
tvals = [currp] #recording of time
pvals = [currp] #recording of proteins
#end of part 1 - initial conditions
```

With this in hand, now start simulating:

```
# Part 2 - Stochastic simulation
while currt<tmax:
    # Calculate rates
    decayrate = alpha*currp
    prodrate = beta
```

```
total_rate = decayrate+prodrate

# Find waiting time
deltat = np.random.exponential(1/total_rate) # Find next event
currt = currt+deltat # Move forward in time
prob_produce = prodrate/total_rate # Calculate probability of production

# Update the state
if (np.random.uniform()<prob_produce): # Choose production at random
    currp = currp+1 # Increment number of proteins by 1
else:
    currp = currp-1 # Decrement number of proteins by 1

# Record the state
tvals.append(currt)  #Store the event time
pvals.append(currp) #Store the protein number
```

That's it! Now, plot a stochastic trajectory using the values stored in `pvals`. Three trajectories are plotted below to show some typical variation in the dynamics. The protein number approaches the expected state value of $p^* = \frac{\beta}{\alpha} = \frac{30}{1}$ and then appears to fluctuate about this value. The inherent noise from the sampling process causes the fluctuations around what would be an otherwise stable equilibrium for the analogous ODE dynamics. Here, the plot also includes the expected dynamics from the ODE model - it is not a coincidence that the ODE solution runs through the stochastic models, as explained in the main text the ODE is the expected mean field (averaged) behavior of the stochastic trajectories.



Using this code, try out a few variations, and try converting it into a single function definition. Hint: you can use the arguments (or initial conditions) `([currt, tmax], p0, beta, alpha)`. With this, try to

- Extend or shorten the maximum time
- Double the production, leaving the decay the same
- Double both the production and decay rates – does the mean change? If not, do you see any difference?
- Explore the system on your own ...

1.4 Bonus: discrete sampling of stochastic trajectories

Comparing stochastic trajectories across time can be difficult because the events occur at random times. It can be useful to sample the trajectory at fixed times. Sampling at fixed times would seem to be intrinsic

Figure 2: Interval sampling of stochastic trajectories, comparing the process (top) with sampled output at 10x per hr (middle) and sample output at 2x per hr (bottom).

to the nature of experimental design, but an event-driven simulation reports back changes in values at exponentially distributed intervals – these are almost certainly not equally spaced! Hence, the next objective is to learn how to write a function to sample a stochastic trajectory at fixed times. This pseduocode may be helpful:

```
Choose vector of times.
pre-allocate a vector the same size as the vector of times for sampled protein numbers
loop over vector of times
find the last protein number at the time less than or equal to the current time
place this value into the vector of sampled protein numbers
end
```

This simple concept is powerful, see for example Figure 2, in which the results of a single stochastic trajectory is contrasted with the sampled version at a frequency of 10x and 2x per hr. Such sampling also makes it possible to compare one trajectory to the next.

Challenge Problem: Discrete Sampling of Stochastic Trajectories

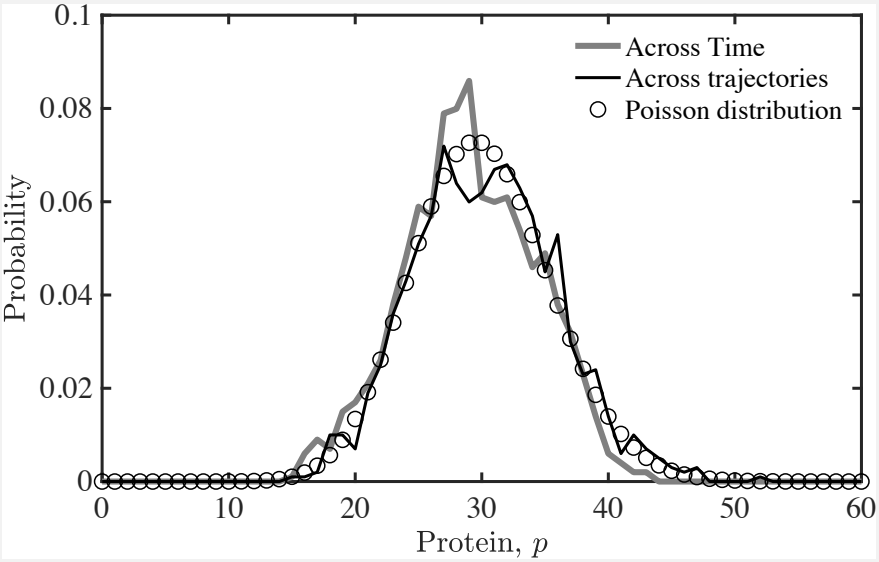
Implement a function to take the output of an event-driven Gillespie algorithm and return the values of the state at discrete or otherwise pre-specified intervals. This will be useful in moving from the system perspective to the measurement perspective – see Figure 2.

1.5 Double bonus: comparing statistics of the 'steady state'

The double bonus – appearing for the first time in this workshop. If you have made it this far, you now have a working Gillespie algorithm and a means to sample these stochastic trajectories at fixed intervals. Now you are ready for the next challenge: to compare statistics of the stochastic protein values across time and across distinct trajectories altogether. To do so, first tun the gillespie algorithm with initially 0 proteins for a long time–say 20 hours. Use the last point as the initial condition for a new simulation. Run the simulation for another 100 hours. Sample this trajectory between 20 and 120 hours for 1001 equally spaced time points (this corresponds to sampling every 6 minutes). What is the mean and variance of the protein numbers across time for this simulation? What is the distribution of protein numbers across time? Now repeatedly simulate the dynamics for 20 hours starting with 0 proteins each time. Do this for 1001 runs. For each simulation store the final protein number. What is the mean and variance of the final protein numbers across the trajectories? Compare this distribution of final protein numbers to the distribution across time that we computed above. The results are remarkably... the same! The reasons why are deep, but reflect the 'ergodicity' of the problem, the fact that a sufficiently spaced sample of the system's steady state is representative of the entire state space. The following challenge problem formalizes this, though given time constraints you might want to only take on one of the two ways of calculating the steady state. The other remarkable thing is that the distribution itself is a Poisson distribution – for reasons why see the main text.

Challenge Problem: Steady State Distributions of Stochastic Gene Expression

Develop code that compares and contrasts the output of the long-term sampling of a single trajectory vs. sampling of many trajectories at distinct points. Use $\beta = 30$, $\alpha = 1$, and sample at the 20 hr point across 1000 runs or sample every 0.1 hrs after reaching the 20 hr point. If your code works it should look like:



Notice, the distributions are nearly the same. The notion that the state fluctuates across time in the same manner that it fluctuates across replicates is referred to as ergodicity. The main take-away of ergodicity is that if the dynamics are not expected to change significantly an experimentalist can choose to obtain statistics by either repeating an experiment or by increasing the number of samples of a trajectory. This is not expected to hold when transients are a major factor; hence, we simulated 20 hours after the initial condition before taking data.

2 Loading and saving data

Loading and saving `.mat` files is easy in Python. First, save your trajectory data

```
from tempfile import TemporaryFile
#save protein trajectory data
pfile = TemporaryFile()
np.save(pfile,pvals)
#load it into a new variable
newTraj = np.load(pfile)
```

In addition, a common data format is `.csv`. Reading `.csv` is done in Python using the `np.genfromtxt` function. First, create a data file in which the first row has a header row and then make up a few rows of data. Name the data `fakecsvdata.csv`. Next, try loading `fakecsvdata`.

An error gets thrown up because there are non-numeric entries in the csv. Instead read the data starting from the second row, i.e.,

```
x=np.genfromtxt('fakecsvdata.csv',delimiter=',',skip_header=1)
```

where `skip_header` is the number of rows to skip.

3 Take-away Points

The major take-away points from this workshop include

- Random events that occur at each moment with the same rate independent of previous events are term Poisson processes.
- The time between events in a Poisson process are distributed exponentially.
- The average time between events in a Poisson process is equal to the inverse of the rate.
- When multiple events take place, the average time to the next event (of any kind) is equal to the inverse of the *sum* of the underlying rates.
- The Gillespie algorithm represents event-driven dynamics, where the time between events is exponentially distributed, then the state is updated given the event type, and the process repeats
- A stochastic simulation of protein dynamics recapitulates the deterministic model, on average.
- The steady state of a stochastic simulatioin of protein dynamics is a random state variable whose value is distributed like a Poisson distribution.
- The Gillespie algorithm can be extended to multi-dimensional state spaces and many types of processes using the same core techniques.

4 BONUS PROBLEM: Bistability and Noise

For this problem, we will look at a bistable system with noise. The differential equation below represents a system with an autoregulatory feedback loop where X activates itself. (Θ is a step function, which outputs 1 if its input is greater than 0, and 0 otherwise.)

$$\frac{dX}{dt} = \beta_+ \Theta(X - K) + \beta_- \Theta(K - X) - \alpha X \tag{2}$$

We can think of this equation as representing the production of a protein. The more protein is produced, the more will be produced. There is a critical value of β_+ above which the system will exhibit bistability in the long term, that is, it will have an "ON" state (where it produces a certain maximal level of protein) and an "OFF" state (where it produces the basal level of protein), depending on the starting concentration of protein.

Parameters of the system:

- dilution rate α, which represents how quickly proteins degrade
- max production rate β_+
- basal level of production β_-
- half-saturation constant K

In this problem, you will simulate a stochastic version of the dynamics represented by this equation using the Gillespie algorithm. We will assume $\alpha = 1$/hr, $\beta_- = 20$ nM/hr, and K = 30 nM. **Optional:** use the above equation to show that the critical value of β_+ is 30 nM/hr.

4.1 Using the Gillespie Algorithm to Simulate Positive Feedback

Modify the Gillespie algorithm that you wrote earlier to simulate stochastic gene expression given the autoregulatory positive feedback loop described above.

Use max production rates $\beta_+ = 35$, 40, and 50 nM/hr, plot examples of trajectories that start from the "OFF" and "ON" states. (Hint: think about what it means for the system to be "OFF" or "ON", and how that relates to the maximum and basal production rates.)

4.2 How Long Before the System Turns "ON"?

Now, we will initialize the cell in the "OFF" state, and characterize the time it takes to move to the "ON" state. (Hint: You will need to run many simulations and store how long it takes to reach the "ON" state each time.)

Plot a histogram of the times that you found. What does the distribution look like? You may also want to look at more sample trajectories to get an idea of what the system is doing.

4.3 How Long Before the System Turns "OFF"?

Now we will do the same thing we just did, but starting from the "ON" state this time. Initialize the cell in the "ON" state and characterize the time it takes to move to the "OFF" state.

Again, plot a histogram of the times that you found. What does the distribution look like? Is it different from the distribution of "OFF"→"ON" times?

4.4 Distribution of Protein Concentration Over Long Times

Finally, run a simulation for a very long time. Plot a histogram of protein concentration during this simulation. Can you see bistability in action in this plot?

Bonus: How frequently and for how long do you have to sample cells to ensure that you 'see' the bistability in action? (Hint: refer to the section above on sampling from trajectories.)

5 Solutions to Challenge Problems

This section includes solutions... try to figure this out on your own before you look! Really! You can do it!

Solutions to Challenge Problem: State-Dependent Rates. The probability of a decay event taking place before a production event is the ratio of the decay rate to the total rate of all processes. Hence, it is $\alpha p/(\beta + \alpha p)$. That is a saturating function of p. Note that at equilibrium then $\beta = \alpha p^*$, such that by definition, the probability of decay is $1/2$, the probability of production is $1/2$, and so the system is equally likely to increase protein levels by 1 as it is to decrease protein levels by 1. The visualization of the probability of decay is enabled by the following code:

```
# Set rates and probability of decay
alpha = 1
r = 30
probdecay = lambda p: alpha*p/(r+alpha*p)
# Plot the probability of decay as a function of proteins
pvec = np.arange(300)
plt.plot(pvec,probdecay(pvec),color='k',marker='.',markersize=10,linestyle='')
plt.ylim(0,1)
```

Solutions to Challenge Problem: Discrete Sampling of Stochastic Trajectories. There are many ways to accomplish this goal. Below is a function which implements the pseudocode described in the laboratory text. It takes as input the observed time and state values, `t` and `y`, respectively, as well as a specific range of times `trange`. It then returns the time In essence, it moves from one sample time point to the next and then scans through the event times until the event time exceeds the sample time. At that point, the value of the state is saved, and then the loop advances. This function retrospectively samples times and states, irrespective of the event timings returned by the Gillespie algorithm. Note that a full code would also deal with the possibility of boundary checking, i.e., ensuring that the sample times and event times are bounded.

```
def stochsim_protein(tlim,p0,beta,alpha):
    '''Uses the Gillespie Algorithm to stochastically simulate protein
    production.
    tlim is a 2 element object with the start and stop time
    p0 is initial protein count
    beta is production rate
    alpha is decay rate'''
    #stochastic gene expression
    #part 1 - initial conditions
    currp = p0 #current proteins
    tmax = tlim[1] #max time
    currt = tlim[0] #current time
    tvals = [currt] #recording of time
    pvals = [currp] #recording of proteins
    #end of part 1 - initial conditions

    # Part 2 - Stochastic simulation
    while currt<tmax:
        # Calculate rates
        decayrate = alpha*currp
        prodrate = beta
        total_rate = decayrate+prodrate

        # Find waiting time
        deltat = np.random.exponential(1/total_rate) # Find next event
        currt = currt+deltat # Move forward in time
        prob_produce = prodrate/total_rate # Calculate probability of production

        # Update the state
        if (np.random.uniform()<prob_produce): # Choose production at random
            currp = currp+1 # Increment number of proteins by 1
        else:
            currp = currp-1 # Decrement number of proteins by 1

        # Record the state
        tvals.append(currt)  #Store the event time
        pvals.append(currp) #Store the protein number
    return tvals, pvals
```

Solutions to Challenge Problem: Discrete Sampling of Stochastic Trajectories (cont).

```
def sample_traj(t,y,trange):
    """Samples a trajectory t,y at discrete intervals"""
    ts = np.zeros(np.shape(trange))
    ys = np.zeros(np.shape(trange))
    #initialize
    curt = trange[0]
    ind=np.where(t<=curt)[0]
    curind = ind[-1]
    ts[0]=t[curind]
    ys[0]=y[curind]

    #Scans across the sample interval and then moves the
    # actual dynamics forward until we cross it
    for i in range(1,len(trange)):
        nextt=trange[i]
        while t[curind]<nextt:
            curind=curind+1
        ts[i]=nextt
        ys[i]=y[curind-1]
    return ts, ys
```

With this code in place, a comparison can be made as follows

```
# Run the Gillespie algorithm
alpha=1
beta=30
tmax=2

# Plot the original
t,y = stochsim_protein([0, tmax],0,beta,alpha)
tmph=plt.plot(t,y,'ko','grey')

# Sample the trajectory every 0.1 hrs
ts,ys=sample_traj(t,y,np.linsapce(0,tmax,0.1))
# Plot the sampled trajectory
tmph=plt.plot(ts,ys,'ko-',linewidth=3)
```

Solutions to Challenge Problem: Steady State Distributions of Stochastic Gene Expression. The solution to this challenge problem involves running stochastic trajectories, sampling, saving, and repeating. The following code snippet provides sufficient detail to reproduce the main results:

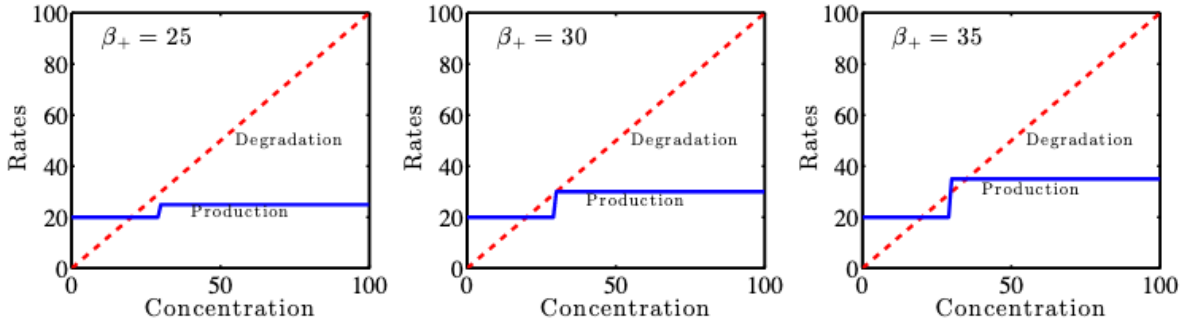
```
# Parameters
beta = 30
alpha = 1

# Long term trajectory
t, y = stochsim_protein([0,20],0,beta,alpha)
y0 = y[-1]
t, y = stochsim_protein([0,100],y0,beta,alpha)
trange = np.linspace(0,100,1001)
ts, ys = sample_traj(t,y,trange)

# Histogram
py,px = np.histogram(ys,np.arange(0,61))
plt.plot(px[:-1],py/sum(py),linewidth=4,color=[0.5,0.5,0.5])

# Simulate many trajectories
numsamps = 1001
final_y0 = np.zeros(numsamps)
for j in range(numsamps):
     t, y = stochsim_protein([0,20],0,beta,alpha)
     final_y0[j] = y[-1]

py2,px2 = np.histogram(final_y0,np.arange(0,61))
plt.plot(px2[:-1],py2/sum(py2),linewidth=2,color='k')

# Calculate the distribution in theory
px_theory = np.arange(0,61)
py_theory = stats.poisson.pmf(px_theory,beta/alpha)
plt.scatter(px_theory,py_theory,c='w',edgecolors='k',s=100,linewidth=2)

#Additional plot labels at your discretion
```

Bonus Problem Solution: Bistability with Noise

Recall that the dynamics can be written as:

$$\frac{dX}{dt} = \beta_+\Theta(X-K) + \beta_-\Theta(K-X) - \alpha X \tag{3}$$

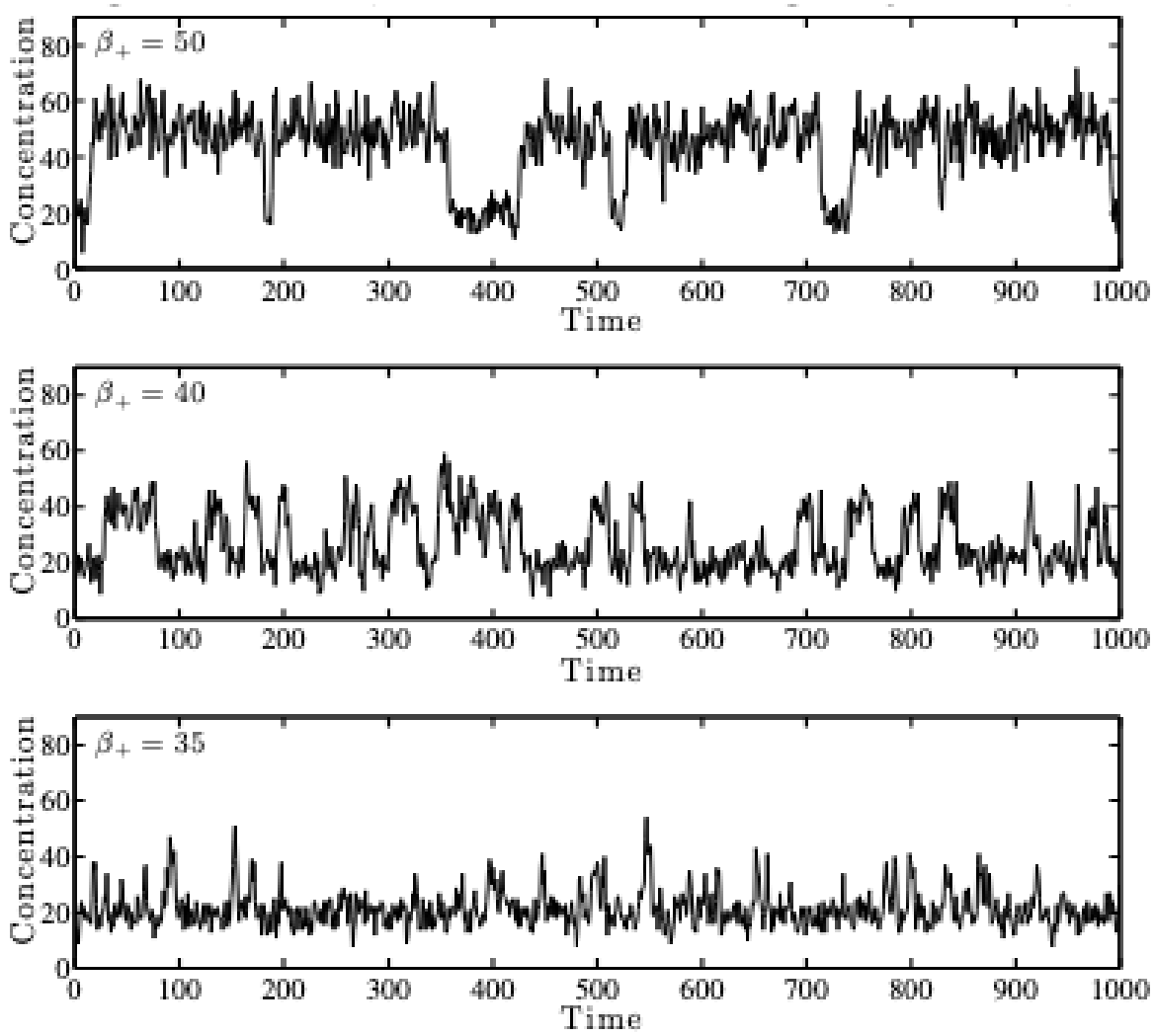
where the Θ function is 1 when its argument is positive and 0 otherwise. Hence for bistability the density-dependent production and density-dependent degradation must cross more than once, i.e., $\alpha K > \beta_-$ and $\alpha K < \beta_+$. Here, given $\alpha = 1$, this implies that the first condition is satisfied, but the second condition is only satisfied when $\beta_+ > 30$. In other words, there must be at least a difference of 10 nM/hr of production in the on versus the off state, as is apparent here. In the plot below each crossing of the blue (production) curve with the red (degradation) curve denotes a fixed point. When $\beta_+ = 35$ there are 3 fixed points, at $X^* = 20$, 30 and 35, the first and last of which are stable and the middle of which is unstable.



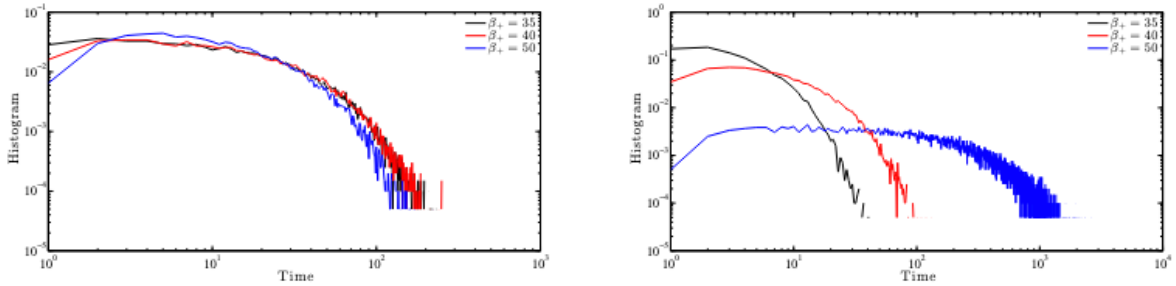
Next, the following code snippet captures the key ideas of the stochastic model in which there is degradation and production, the combined rates set the interval between events, and the relative rates set the probability of selecting one of the events. Here `tcur` refers to the current time and X refers to the concentration. The final steps store the entire trajectory.

```
def positive_feedback_gillespie(pars, tf, X=20):

    t = []
    xval = []

    tcur = 0
    while tcur < tf:
        drate = pars["alpha"]*X

        # Calculate next event
        drate = pars["alpha"]*X
        brate = pars["betap"]*(X >= pars["K"]) + pars["betam"]*(X < pars["K"])
        totrate = drate + brate
        dt = -1/totrate*np.log(np.random.random())
        tcur += dt

        # Event type
        if (np.random.random() < (drate/(drate+brate))):
             X -= 1
        else:
             X += 1

         # Update events
         t.append(tcur)
         xval.append(X)

     return t, xval
```

Using this model, we find the following trajectories, using $\beta_+ = 35$, 40 and 50 respectively:



As is apparent, as the ON state increases in concentration, the system tends to spend ever more time in the ON state before switching back. Similarly, with these results in hand, we initialize many simulations in OFF and ask the question: how long does it take for the system to first reach a concentration with $X(t) = \beta_+$? We find that the mean time increases with β_+ such that the mean is 22, 26 and 28 hrs for $\beta_+ = 35$, 40 and 50 respectively. Yet this time has largely to do with the time it takes to get to the ON state, but not necessarily to cross the barrier at $X = K$. Notice however that the difference is far more stark when we begin in the ON state, particularly when the ON state is well separated from the value K. In this case, the mean is 5, 13, and 270 hrs for $\beta_+ = 35$, 40 and 50 respectively. In fact, there are very long tails as is apparent in the right-most histogram below. Hence, the ON state has long memory, but the OFF state is susceptible to fluctuation-induced reversal. A critical point here is that these are NOT exponential distributions. Because the system cannot jump across theese barriers at once, this is equivalent to first-passage time problems, often which yield an 'inverse Gaussian' distribution with a mode and a long tail.



Finally, setting $\beta_+ = 40$ nM/hr, one can sample over a long trajectory of 5000 hrs and averaging the densities. The mean time to go from OFF to ON is 26 hrs and from ON to OFF is 13 hrs. Hence, it would seem that the rate of switching form ON to OFF is approximately 1/2 that of OFF to ON and therefore that one would expect approximately 2-fold more cells in the OFF state than in the ON state. In fact, I find that there are 32% of cells in the ON state ($X > 30$) and 68% of cells in the OFF state ($X < 30$), which is a ratio of 2.12 — not bad! That is, we can think of the fluctuations in terms of effective transition rates between just 2 macroscopic states despite the complexity of the microscopic model. As such, sampling duration should exceed the sum of residence times and sampling intervals should be significantly less than the transition times to resolve switches between the states. The figure below includes all three distributions, from left-to-right it is evident that the ensembles moves from largely OFF to largely ON.