May 15, 2018 – QBioS Hands-On Modeling Workshop – Evolutionary Dynamics
Prof. Joshua Weitz

Would you like to be able to simulate evolutionary dynamics? Building upon yesterday's hands-on intro to basic programming skills, we will get to the point where you can do it – in this very session. The goal of this lab is to understand how to:

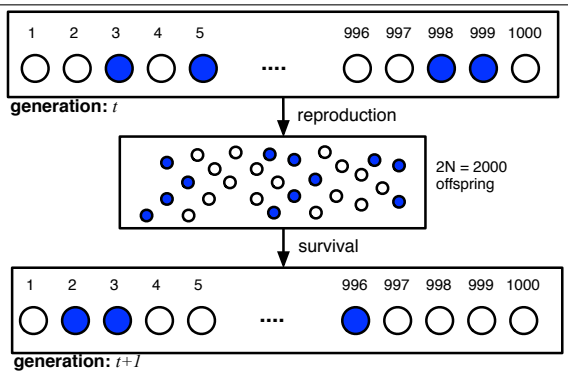
- Translate models with stochastic transitions between states.
- Simulate genetic drift in a finite sized population.
- Simulate evolutionary dynamics, including the effects of beneficial mutations, that can be compared to measurements.

To get there, the lab focuses on computational approaches for simulating 'Markov' processes. A Markov process denotes a set of probabilistic rules governing how a system behaves. Markov processes have a special feature: stochastic changes in the state of the system depend on the *current* state and not on prior states. For example, to simulate the dynamics of a cell as a Markov process requires that we specify the abundance of RNA, proteins, etc. at a given moment. These abundances then determine the probability of subsequent changes, whether of RNA transcription, proteins translation, receptor-ligand binding, and so on. The same principle applies to simulations of populations. For example, to understand the frequency of strains at generation $g + 1$ in an evolutionary model requires that we specify the rules for changes in the frequency of strains given those present at generation g. In essence, Markov process are 'memoryless'. Although real biological systems can have memory, a Markov process would increase the number/complexity of states at a given moment in time to represent whatever 'memory' the system has. These concepts may seem needlessly abstract, but they enable the following simple, conceptual framework for building models of living systems:

1. **Initialize state** Set the initial conditions for the model.
2. **Determine the transition probabilities** Based on the current state of the system, determine the probabilities that the system moves to a different state or remains in its current state.
3. **Update the system state** Change the state of the system in a stochastic fashion (i.e., by change) weighted by the transition probabilities.
4. **Return to step 2 and repeat**

This kind of Markov process can be applied in many contexts. Today, we will aim to build towards a specific simulation target: evolutionary dynamics of bacterial populations. The basis for our work will be the slow and steady development of what is widely known as the 'Wright-Fisher' (WF) model of population genetics (see Figure 1). This model has just a few ingredients – reproduction and survival – as applied to finite population. Without any other ingredients, these two elements alone can give rise to a number of interesting phenomena. And, with just a bit of work, we can extend the WF model to include selection, mutation, and connect it directly with experimental evolutionary data. Onwards!

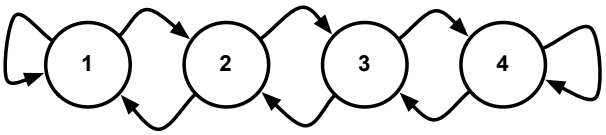
Figure 1: Schematic of the 'Wright-Fisher' model of population genetics. Consider a population with $N = 1000$ individual bacterial cells in a flask. Each bacteria divides, yielding $2N$ cells. The system is then diluted into fresh media so that only 50% of the cells 'survive'. Hence after one 'generation', the system still has N cells. In that case, the process of birth and death can lead to changes in the frequency of genotypes – and inclusion of mutation can lead to introduction of new genotypes altogether.



1 Transition matrices in Markov Processes

1.1 Some preliminary definitions

Markov chains are a special kind of Markov process where discrete transitions between states of a system occur stochastically. The transition matrix dictates the probability of a transition between any two states. For example, the graphic below denotes a system with four states (labeled 1, 2, 3, and 4).



The arrows denote the possible transitions, i.e.,

From state 1 Either to remain in state 1 or to switch to state 2

From state 2 Either to switch to state 1 or to state 3

From state 3 Either to switch state 2 or to state 4

From state 4 Either to switch to state 3 or to remain in state 4

In order to finalize our description of the system's dynamics we must also specify the probabilities that a system will make a transition between state i to another state j. As part of this example, assume that all transitions are weighted equally, i.e., occur with a probability of 0.5 (i.e., 50%). Hence, we can represent the transition probabilities as a 4×4 matrix, T:

		Current State			
		1	2	3	4
Next state	1	0.5	0.5	0	0
	2	0.5	0	0.5	0
	3	0	0.5	0	0.5
	4	0	0	0.5	0.5

Here, the element, T_{ij} is the transition probability from state j to state i, e.g., the first column refer to transitions from the 1 state. A few points should be apparent:

- All entries must be non-negative (i.e., zero or greater).
- The sum of entries in a column must equal 1.

Later, we will show that the sum of entries in a row need not equal 1. The rationale is that the vertical entries denote transitions from the state in the column to some state in the rows. Because the system must end up in one particular state, then the total probability must equal 1. First import `numpy`, `matplotlib` and `stats`, we will use `stats` to do the linear regression (fitting first order polynomial)

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
```

To start, write your own code to store the transition probabilities in a variable `Tmat` – denoting the transition matrix, so that it returns the following:

```
>> Tmat
       array([[ 0.5,  0.5,  0. ,  0. ],
              [ 0.5,  0. ,  0.5,  0. ],
              [ 0. ,  0.5,  0. ,  0.5],
              [ 0. ,  0. ,  0.5,  0.5]])
```

Recall that Python uses an index notation (index starts from zero) where rows come first, i.e.,

```
>> Tmat[1, 2] = 0.9
```

would assign the value 0.9 to the 2nd row and 3rd column entry of the matrix `T`. When ready, have an instructor stop by so we can give you a ✓.

1.2 Simulating a Markov process

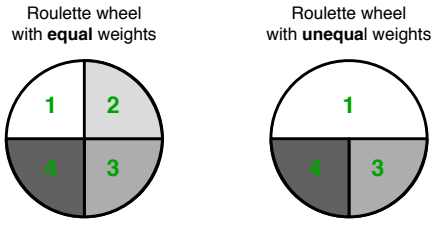
With the transition probabilities in place, it is time to simulate the Markov process. Consider a system that starts in state 1. What could happen next? For example, only one of the following is a viable sequence (can you identify it?):

- 1, 1, 2, 3, 4, 3, 4
- 1, 2, 3, 3, 2, 3, 2
- 1, 2, 3, 4, 4, 3, 1

Your next challenge: write a program to simulate dynamics for 50 steps starting at an initial condition $X_{t=0} = 1$. The following code snippet will be helpful.

```
Tmat = fill in the transition matrix
X0 = 1;    # Initial condition
numgen = 50;    # Number of generations
stochtraj = [X0]  # Initiate the trajectory
currX = X0  # Current value
for g in range(numgen):
    p_trans = Tmat[:, currX] # Transition probabilities in the currX column
    nextX = randi_wheel(p_trans) # Choose a new state based on probabilities
    stochtraj.append(nextX) # Save the state to the trajectory
    currX = nextX # Update the current state
plt.plot(range(numgen + 1),stochtraj,'bo-', linewidth=1, markersize=8)
plt.xlabel('Time',fontsize=20)
plt.ylabel('State',fontsize=20)
plt.show()
```

All looks fine here, except for one part. How do we choose the next state without a series of `if`, `then`, and `else` statements? It may help to recall a roulette wheel:

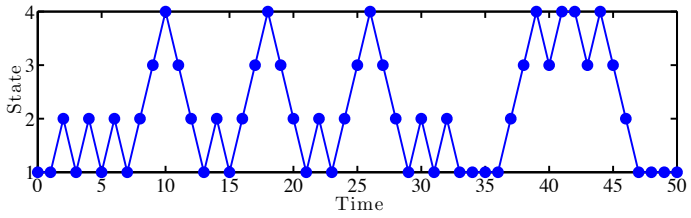


The purpose of the function `randi_wheel` is to take a vector that sums to 1, and return a random index to a state weighted by the values in the vector. In fact, even if the input doesn't sum to one, then we can divide by the weights to ensure probabilities. The algorithm itself draws a random number between 0 and 1 and then checks which state it lands in, i.e., weighted by the wedges of the biased roulette wheel.

```
def randi_wheel(weights):
# function x = randi_wheel(weights)
# Returns a single index returned to the state selected at
# random specified by the non-negative values in weights.
# The function will return a -1 error code if the weights
```

```
# include negative values
    if sum(weights < 0): # check for negatives, 'weights < 0' returns logical array
         x = -1
    else:
        tmp_weights = weights/sum(weights) # normalize
        tmp_order = np.cumsum(tmp_weights) # take a cumulative sum
        x = np.argmax(tmp_order >= np.random.rand(1)) # draw a random number
    return x
```

Challenge: update the code and plot the dynamics. A typical series should look something like this:



Try this a few times. Each stochastic realization is different. Some plotting commands maybe helpful

```
plt.plot(range(numgen + 1),stochtraj +
         np.ones([numgen + 1]),'bo-', linewidth=1, markersize=8) # state space {1,2,3,4}
plt.xlabel('Time',fontsize=20)
plt.ylabel('State',fontsize=20)
plt.show()
```

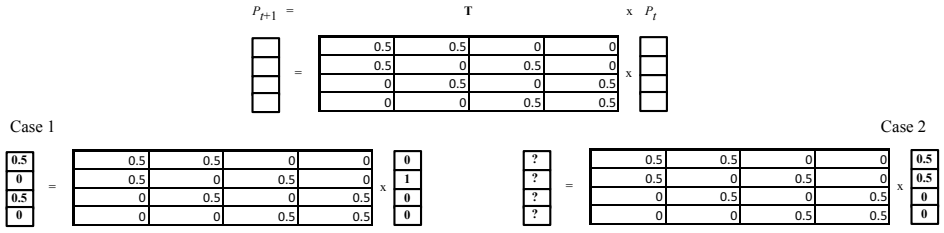
At this stage you've completed your first major checkpoint: a simulation of a stochastic random process. We just need to get one more concept in place before moving to the 'Wright-Fisher' model: how to make predictions that work for an *ensemble* of stochastic realizations, leveraging the transition probabilities embedded in the transition matrix, T.

1.3 From realizations to the ensemble

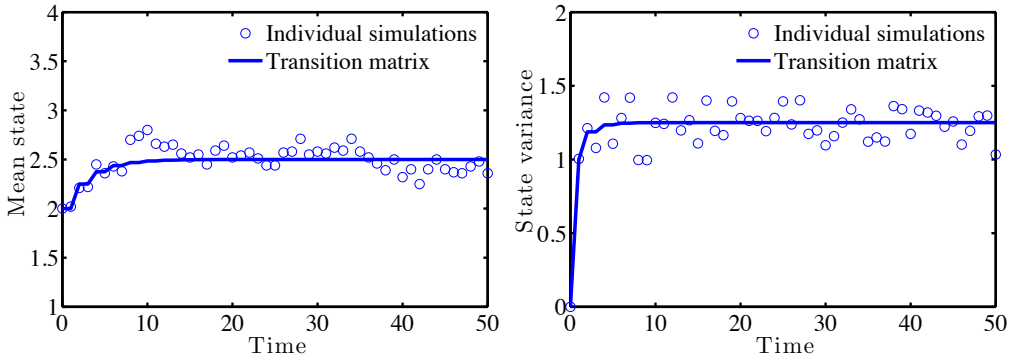
It is not possible to predict the precise trajectory of any one realization of the Markov chain. Instead, we can make accurate predictions of the average (or even higher order properties) of an ensemble of realizations. An ensemble is a collection of many trajectories, each of which share the same transition matrix, T. For example, let us denote $\vec{P}_t$ as the probability that the system is in one of the states at time t. The transition matrix holds the key to predicting the probability that the system is in each of the possible states $\vec{P}_{t+1}$ at time $t+1$. For example, in our original example, if the system starts in state 2 at time $t = 0$, then it has an equal chance to transition to state 1 or 3 at time $t = 1$. Hence, we'd expect that the probability $\vec{P}_1 = [0.5\ 0\ 0.5\ 0]$. In general, the transitions in the Markov chain may be written as follows (see sketch below for a visual representation of the equation, **and a built-in challenge**):

$$\vec{P}_{t+1} = T\vec{P}_t$$

where $\vec{P}_t$ is a vector of probabilities associated with each state at time t.



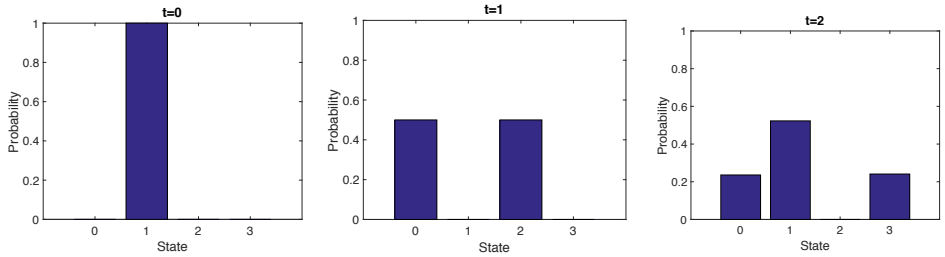
Now, repeat the simulations 100 times and store the ensemble of dynamics. Plot the mean value of the state over time. In addition, plot the variance of the value of the state over time. Note: if the rows of your data array refer to time then use `np.var(mydata,1)` where `mydata` is your data array. See if you can reproduce something like the following sets of images:



Could you have 'predicted' the solid blue line corresponding to the dynamics simulated via the transition matrix without running 1000s (or more) of parallel simulations? Yes, you could have! In fact, the 'theoretical' prediction matches the results from averages of individual simulations. The theoretical probability distribution, mean, and variance over time can be determined from repeatedly applying the transition matrix.

```
nvec = np.array(range(4))
X0vec = np.array([0.0, 1.0, 0.0, 0.0]) #Initial distribution
theormean = np.zeros([numgen + 1, 1]) #Mean state
theorvar = np.zeros([numgen + 1, 1]) #Variance in state distribution
theorp = np.zeros([numgen + 1, 4]) #Theoretical probability vector
Xcur = X0vec #Initializing current state
theormean[0] = X0vec.dot(nvec) #Dot product initial distribution and state numbers (mean state number)
theorp[0, :] = X0vec #Initializing probability vector
for gg in range(numgen):
    Xcur = Tmat.dot(Xcur) #Multiplying distribution by transition matrix
    theormean[gg + 1] = Xcur.dot(nvec) #Updating the mean state value
    currsecondmom = Xcur.dot(np.power(nvec, 2)) #Second moment (for calculating variance)
    theorvar[gg + 1] = currsecondmom - theormean[gg+1]**2 #Calculating variance
    theorp[gg +1, :] = Xcur #Updating probability vector
```

Work through the code, line by line and plot the theoretical curves over your calculated mean and variance from before. Do they agree? Using the transition matrix, we can also visualize the evolution of the entire distribution over time. Next, write code to show the initial evolution of the distribution by plotting the initial distribution and the distribution after one step and then two (hint: use the `bar` command to visualize the distribution), it should look like:



Some useful commands for bar plotting:

```
plt.subplot(1, 3, 1)
plt.bar(range(4), ... ) # the state distribution at some time
```

```
plt.xlabel('State')
plt.ylabel('Probability')
plt.title('t = 0')
plt.xticks(range(4), ('1', '2', '3', '4'))
```

Challenge: dynamics towards an 'absorbing' state. In this last section, your goal is to modify the transition matrix so that

```
Tmat =
      array([[ 1. ,  0.5,  0. ,  0. ],
             [ 0. ,  0. ,  0.5,  0. ],
             [ 0. ,  0.5,  0. ,  0.5],
             [ 0. ,  0. ,  0.5,  0.5]])
```

Before implementing this change, ask yourself: what do you expect to happen over the long term? To visualize the dynamics, write a function that plots the distribution $\vec{P}$ for a given t. You can observe the evolution of the distribution by looping through time, plotting the distribution. In Python, one way you could do that, importing the `FuncAnimation` method of `animation` module, it takes some time to generate the movie (real-time plot). Alternatively, you can use the `subplot` command above to plot the first couple of timesteps, or plot iteratively using a `for` loop. For example, if your data is stored as `theorp`, then you can watch convergence to an equilibrium distribution over time. **Should the probability be equal for all states, or something else?** If you have time, run the stochastic model 100 times and store the ensemble of dynamics. Plot the mean value of the state over time and compare with the theoretical expectation.

2 Wright-Fisher model

It is time to leverage your new-found skills to build the foundational model of evolutionary dynamics: the 'Wright-Fisher' model. Again, the WF model represents dynamics of individuals in a population of fixed size N given non-overlapping generations. In each generation, individuals can reproduce and die. In addition, although individuals may be distinguished by their genotypes or alleles, we will assume at first that all reproduction and survival rates are equal. This means we are building a *neutral* model of evolution. Even though the model is neutral, evolution will still happen! Recall the definition introduced yesterday:

> Evolution: heritable changes in the frequency of genotypes across generations.

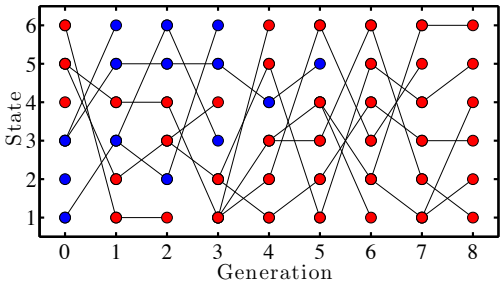
First, if there had been pre-existing variation, e.g., different genotypes, then the frequency of each genotype is unlikely to be exactly the same. Second, it is also possible that some of the daughter cells may be mutants. In that case, the process of birth and death (by removal) would lead to changes in the frequency of genotypes, These steps: birth, death, and mutation are sufficient to provide the foundations of the classic model of population genetics: the Wright Fisher model. And, with a few additions, we'll be able to generalize this model so as to simulate evolutionary dynamics in regimes relevant to studies of experimental evolution of bacteria and yeast, including cases where individual cells vary in their fitness.

2.1 Neutral model of evolution, single locus, two alleles

A simple Wright-Fisher model treats an evolving population with a single gene locus with two alleles, A and B, with non-overlapping generations. In short, the population after each generation is obtained by randomly sampling individuals from the previous generation to reproduce. We treat the case where the fitness difference between alleles is neutral. That is any one A individual has the same probability as any B individual. By studying this model, we can learn about the effect of stochasticity in driving dynamics of evolving, finite populations. First, we will look at some realizations of the dynamics. We start with a population of size $N = 100$, in which half are A alleles. Given neutral drift alone, to what extent does the number of A allele changes after $g = 100$ generations. There are a few ways to do this. As in the illustrative section, we will first develop an individual-based approach and then use a transition matrix approach to compare our ensemble data.

2.1.1 Individual realizations, version 1

Consider a population of size N, and recall the schematic on the first page. You may notice, and perhaps already realized, that the parent of individuals in generation $t+1$ are random. That is to say that one way to simulate WF dynamics is to select the parent of each new individual in generation $t+1$ at random from those parents still present in generation t. As a consequence, the dynamics are: [1]



This simulation result has many features worth noticing. For example, ask yourself the following:

- Do all parents have offspring in the next generation?
- Do all surviving offspring in the last generation have different parents from the founding generation?
- What is the chance that a given parent is not chosen to be have an offspring in the next generation?

[1] Note that plotting the trace-back networks brings up the concept of the 'coalescent', a bit too advanced for the current lab.

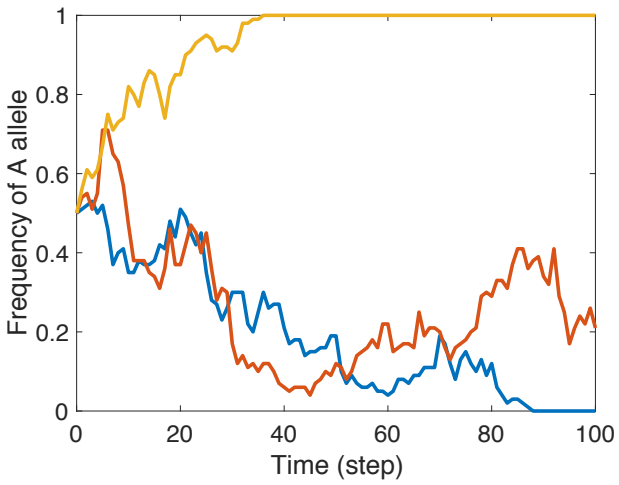
Challenge: modify the code below to develop your initial WF model

```
N = 100 # Individuals
numA = 50
numgen = 100
x = np.zeros([numgen + 1, N])
x[0, 0:numA] = 1 # A allele, type 1
x[0, numA:] = 2  # B allele, type 2
fracA = np.zeros([numgen + 1, 1])
fracA[0] = numA/N

for g in range(numgen):
    for i in range(N): # Identify parent of each new individual
        tmpid = ... # Find the parent, at random
        x[g+1, i] = ... # Assign the same type to the offspring as the parent
    fracA[g+1] = sum(x[g+1, :] == 1)/N

plt.hold = True
plt.plot(range(numgen + 1),fracA, linewidth=2) # Plot the trajectory
plt.xlabel('Time (step)',fontsize = 20)
plt.ylabel('Frequency of A allele',fontsize = 15)
plt.show()
```

If you have this running successfully, then modify the code so that you can visualize multiple trajectories at once. You might find that some of the trajectories reach the 0 or 1 state – this is called an absorbing state, because once the Markov chain reaches there it cannot get out!



2.1.2 Individual realizations, version 2

In this next approach, we will ignore the microstates by leveraging the symmetry inherent in our use of a neutral model. Each individual is the same, the difference is only that the fraction of individuals of type A is potentially different from one generation to the next. Hence, the process in version 1, above, is equivalent to *sampling from a binomial distribution*. That is, there are N offspring (think of them as 'trials'). For each offspring, there is a $p = n_A/N$ chance that the parent is of type A given that there are n_A individuals of

type A in generation t. The probability of selecting k individuals of type A is then

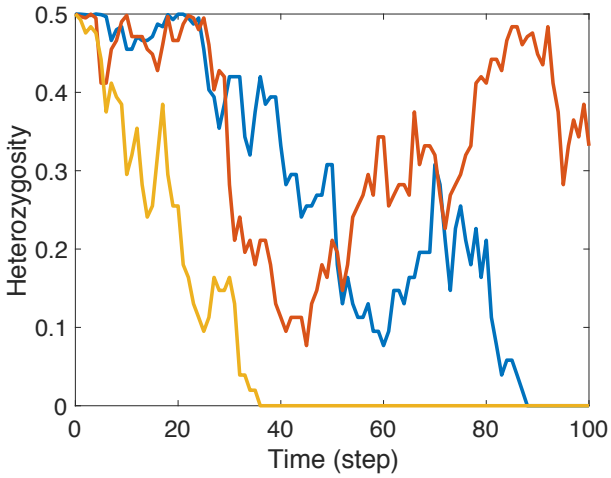
$$p(k) = p^k (1-p)^{N-k} \left(\frac{N!}{k!(N-k)!} \right) \tag{1}$$

In other words, a type A individual is selected exactly k times each with probability p, multiplied by the selection of exactly $N-k$ individuals of type B, and finally multiplied by the number of permutations of choosing k individuals out of N (i.e., 'N choose k'). In Python, it is possible to sample from a binomial distribution using the `np.random.binomial` function, where the code snippet below is left for you to fill in one component (you can use `np.lookfor` to find the documentation if you are unsure of inputs):

```
N = 100
newA = int(N/2)
newA = np.random.binomial( ... )
```

Adapt that code to loop for $g = 100$ generations, storing the state of the population over each generation. Plot the number of A individuals over time. Did the population reach an absorbing state?

With his model in hand, we can analyze properties of the evolutionary dynamics, e.g., heterozygosity. Heterozygosity measures the diversity of alleles in the populations. Mathematically, heterozygosity is $H = 2p_A p_B = 2p_A(1-p_A)$, where p_A is the fraction of the populations with A allele and, similarly, p_B is the fraction of the population with B allele. Plot the heterozygosity for your trajectory. Does the heterozygosity ever increase? Where is the heterozygosity a maximum? The the corresponding heterozygosity dynamics from the previous 3 trajectories are shown below.

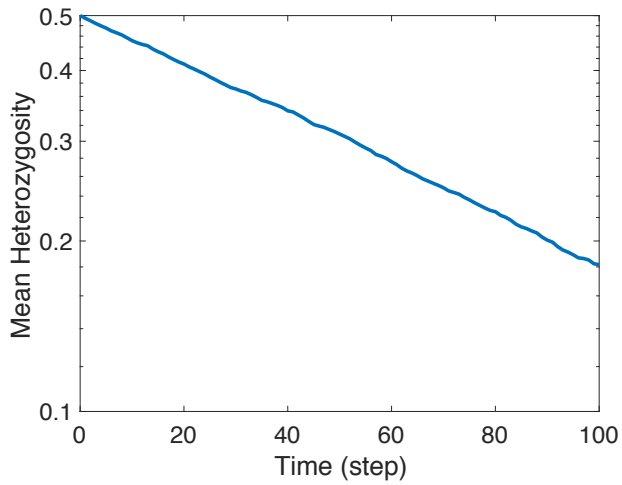


Notice, the heterozygosity is 0 at the absorbing state.

To see how heterozygosity changes on average, repeat the simulation 1000 times. For each run of the simulation, store the dynamics of the heterozygosity (each run contributes a vector with length $g = 100$). Plot the dynamics of the mean value of the heterozygosity. Does the mean heterozygosity ever increase? Where is the heterozygosity a maximum. Plot the heterozygosity with a logarithmic y-axis using the `plt.semilogy` function. Fit a line to the log transformed data. What is the value of the slope? How does it compare to $-N^{-1}$? If your mean heterozygosity is stored as `meanheterozyg` then log transform and fit the slope by using:

```
myslope,intercept,r_value,p_value,std_err = stats.linregress(range(g+1),np.log(meanheterozyg))
```

where the `stats.linregress` command outputs the coefficients to fitting a polynomial. The order of the polynomial is set by the last input, in this case 1.



2.1.3 Transition matrix approach

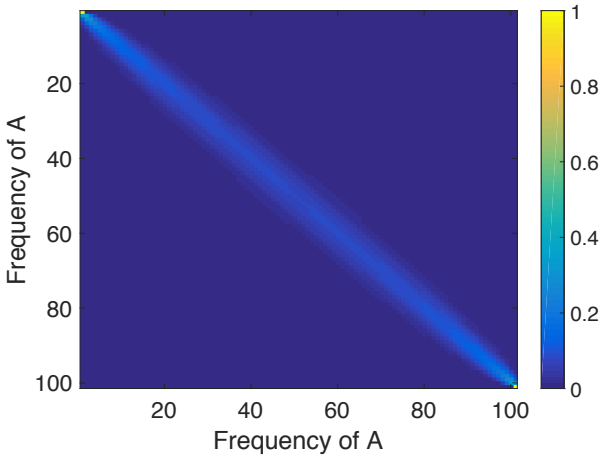
As was hinted at in the prior section, the Wright-Fisher model is an example of a Markov chain. In this case the states are the number of A alleles in the population. The transition probabilities follow the binomial distribution for a given row with probability $p_i = \frac{i-1}{N}$ for the ith row. Construct the transition matrix.

```
N = 100
Tmat = np.zeros([N+1, N+1])
for jj in range(N + 1):
    currp = jj/N
    xvec = np.array(range(N + 1))
    Tmat[:, jj] = stats.binom.pmf(xvec, N, currp)
```

To visualize the transition matrix use the `imshow` command

```
plt.imshow(Tmat, cmap='jet')
plt.colorbar()
plt.xlabel('Frequency of A',fontsize = 15)
plt.ylabel('Frequency of A',fontsize = 15)
plt.show()
```

where the colorbar shows the probability associated with each element of the transition matrix.



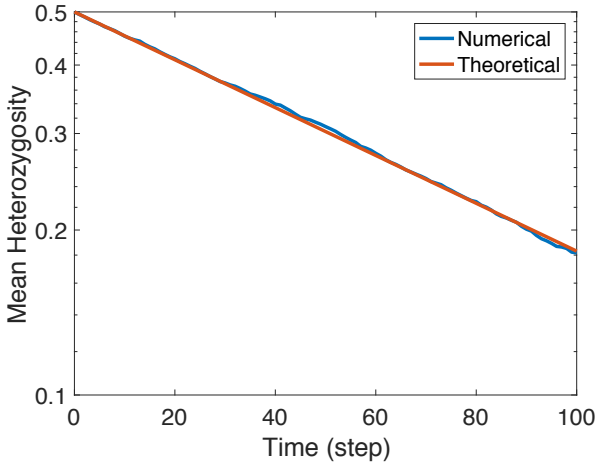
With the transition matrix we can numerically solve for theoretical time evolution of the heterozygosity. This involves first solving for the probability distribution at each time given an initial probability distribution.

```
theorp = np.zeros([N + 1, g+1])
theorp[int(N/2), 0] = 1
for i in range(g):
    theorp[:, i + 1] = Tmat.dot(theorp[:, i])
```

where the time evolution of the probability distribution follows from repeatedly applying the transition matrix to the vector of initial conditions `X0vec`. The mean heterozygosity follows from:

$$\langle H(t) \rangle = \sum_{n=0}^{N} 2\frac{n}{N}(1 - \frac{n}{N})\mathbb{P}_n(t) \tag{2}$$

where $\mathbb{P}_n(t)$ is defined based on the theoretical distribution calculated as `theorp`. Overplot the theoretical and numerical mean heterozygosity dynamics.



3 Clonal interference

If you have reached this point and still want a challenge, consider trying to extend the WF model to include a few more features:

- Two loci, rather than one.
- Denote the first locus as the 'selective' locus. Hence, alleles on this locus have the chance, with some probability to confer a fitness advantage s to the genotype, in which case you will have to use `randi_wheel` to generate biased reproduction.
- Make the second locus neutral, but with the possibility of having an infinite number of alleles.

You will also have to set a mutation rate and consider 'linkage'. Hence, individuals will be characterized by two states and so there can be far more than 2 genotypes concurrent at any point. In this way, neutral alleles can hitchhike with selective alleles, and selective alleles can compete with one another. And, if you really want to get fancy, you can make the selective advantages accumulate with time! Soon, there won't be much difference between your model and the ones you see in Nature, Science, and biorxiv!