Simulation with Python (and NumPy)

In this exercise, you will use NumPy to build a general simulator for the Wright-Fisher model and use matplotlib to plot some simple properties of the evolution. You will simulate N individuals in your population, and in each generation each individual will reproduce with probability proportional to its share of the total fitness. So if individual *i* has fitness s_i, then each off spring has probability $s_i / \sum s$ of descending from individual *i*.

In a new terminal window, change to the `dadiExercise/simulation` directory and edit a new script file: `nano wfsim.py`. This is where you'll write your Python script. Open another terminal window, change to the `simulation` directory, and start iPython with `ipython --pylab`. In the iPython window, you can run your script at any time with `%run wfsim.py`. (Be sure to save your script when you change it!) If your script hangs, use `ctrl-C` to stop execution.

1. We begin our script by importing Numpy and by defining values for our population size and simulation time. We'll start with a very small population and number of generations, for easier debugging.
   ```
   import numpy as np
   N = 5
   generations = 2
   ```
   We'll initialize our population with random fitnesses. Add the following line to your script.
   ```
   pop = np.ones(N) + np.random.uniform(-0.5,0.5,N)
   ```
   Run your script using `%run wfsim.py` in the iPython window. Check that `pop` is a sequence of random fitnesses, with a reasonable range.

2. Now we begin looping for a number of generations. Our first step in each loop is to calculate the proportion of the total fitness possessed by each individual.
   ```
   for gen in range(generations):
       fitnessprops = pop/np.sum(pop)
   ```
   Run your script. Compare `fitnessprops` with `pop` to ensure that the calculation gave what you expected.

3. Next we need to randomly sample the number of offspring for each individual. This is an example of *multinomial* sampling. We'll be sampling `N` new individuals, with `fitnessprops` being the probability of descent from each individual in the previous generation. We can use a function built into Numpy for this. So inside your for loop, add:
   ```
   numoffspring = np.random.multinomial(N, fitnessprops)
   ```
   Run your script. The `numoffspring` array should consist of integers, where the *i*th entry corresponds to the number of offspring for individual *i*. Check that the length of this array, `len(numoffspring)`, and the sum of this array, `np.sum(numoffspring)`, both equal `N`.

4. Next, we need to assemble the population for the next generation. To do so, we'll use a for loop and zip to simultaneously iterate over `pop` and `numoffspring`, so we'll pull out each fitness and the number of offspring that carry it. We'll then assemble the `newpop` list using `extend` commands. (Note that `[val]*ii`, where `val` is any value and `ii` is an integer will create a list of length `ii` that consists of repeated `val` entries.) So inside your for loop, add the lines:

```
newpop = []
for fitness, num in zip(pop, numoffspring):
        newpop.extend([fitness]*num)
```

Run your script. Check that `newpop` is a list of length `N` whose elements came from `pop`.

5. Before we can go back to the top of the loop, we'll need to replace `pop` with `newpop`. We'll also need to convert `newpop` into a numpy array so it can feed into the arithmetic line we have at the top of the loop. To do this, add the line below to the end of your outer for loop. (Be careful that indentation matches your outer loop, not the inner loop.)
```
pop = np.array(newpop)
```

   Run your script. Ensure it doesn't crash with an error.

6. We have a running simulation now, but we're losing all the intermediate results. Let's save them. Before your outer for loop, add the line `history=[pop]` to create a list that starts off holding the initial population data. Inside, but at the end of your loop over generations, append the newly created pop to that history as well.
```
history.append(pop)
```

   Run your script. Check that history has length 3 and that each element is a different array of fitnesses.

7. Next, we'll calculate some statistics about the fitness history of the population. In particular, we'll calculate the mean and standard deviation of the fitness over the generations. To do so, add these lines to the end of your script.
```
mean_fitness = np.mean(history, axis=1)
std_fitness = np.std(history, axis=1)
```
   The axis=1 arguments tell Numpy that we want to take the mean for each row in history, treating it as a 2D array, so we'll have the mean fitness versus time.

   Run your script. Check that `mean_fitness` and `std_fitness` have length 3. Also check that their values make sense when compared with your history.

8. Lastly, we'll use matplotlib to plot our results. We'll create a figure with two subplots, one for mean and one for standard deviation.
```
import matplotlib.pyplot as plt
fig = plt.figure(5, figsize=(3,5))
fig.clear()
plt.subplot(2,1,1)
plt.plot(mean_fitness, '-o')
plt.ylabel('mean fitness')
plt.subplot(2,1,2)
plt.plot(std_fitness, '-o')
plt.ylabel('std. dev. Fitness')
plt.xlabel('generation')
plt.tight_layout()
plt.show()
```

   Edit your script to set `N=100` and `generations=100`. Run it several times. What generally happens to the mean and standard deviation of fitness? Does mean fitness always increase monotonically?