EE194/BIO196: Modeling Biological Systems
HW 3: Population growth with stochasticity

Overview

In this homework, we will take HW #2 problem #1 a bit further. Our use of two arrays to model a population's vital rates will not change. However, this time we will add *stochasticity* – i.e., an element of randomness – into the model. We will also add *quantization*, which is the recognition that in real life you cannot have a fraction of an individual.

Stochasticity

Nature does not work with perfect predictability. Some years are better than others; perhaps due to weather, perhaps due to interactions with a rich variety of other populations and influences. While we cannot (almost by definition) model such unpredictable events perfectly, we can (and often do) add randomness into our models.

We will use the Python function *random.gauss* (*mean*, *sigma*) to create random numbers for us. We give the function the desired mean and sigma (i.e., the standard deviation), and then successive calls to it will return random numbers whose distribution follows the appropriate bell-shaped curve. Python also has many other random-number facilities, some of which we will use later on in the course.

Start with your code from HW #2 problem #1 (the simple population-growth model using 4-element vectors for *m*, *p* and *n*). Use the vital rates from HW #1 problem #1; i.e., m_0=0, m_1=1, m_2=5/6, and m_3=0, and p_0=.6, p_1=.8, p_2=5/6 and p_3=0.

Now modify your simulation to add stochasticity, as follows:

- For each year, use *random.gauss* () to pull a single random number *r*, which will represent how "good" the environment is that year. Always use a mean of zero; the standard deviation will differ in each problem.
- Use this single random number to alter the vital rates. If *r* is, e.g., 0.2, then all vital rates will increase by 20% (i.e., be multiplied by 1.2). Similarly, if *r* is -.3, then all vital rates will decrease by 30% (i.e., be multiplied by 0.7). That is, you simply multiply all vital rates by (1+*r*).
- A bell-shaped curve has tails that extend infinitely far in either direction. However, if *r* is less than -1, you will wind up with negative values for *p* and *m*, and hence negative population numbers, which are clearly unrealistic. Furthermore, if *r* is big enough, you can have values for *p* that are greater than 1, which is equally unrealistic. Thus, you should bound *r* to always be greater than -1, and bound the individual survival rates to always be less than or equal to +1 (e.g., using Python's *numpy.clip*() function).

Quantization

The next thing to do is add *quantization*. All of our previous simulations have allowed fractions of an individual. While this is quite convenient mathematically, it is of course not possible in real life for any given single population. We will use the Python function *numpy.round* (*array*) to round our population numbers to the nearest integer.

You should round to the nearest integer in two places. First, when you use the survival rates *p* to determine how many individuals progress to the next age group, you should round the

numbers. Next, when you use these rounded numbers with *m* to determine the number of new births, you should round the number of new births. As we discussed in class, you do *not* need to round the number of births from each individual age group before adding them (which would prevent you from using a dot product).

While loops

We've already learned how to use *for loops* to run a piece of code a fixed number of times. Problems #2 and #3 of this homework will give us a different goal: running a population simulation until the population dies out, no matter how many years that takes. One common tool to use in this situation is a *while loop*. For example, the code

```
i=0
while (i<5):
        print (i)
        i=i+1
```

will print the numbers 0, 1, 2, 3 and 4. Essentially, it executes the two statements "*print*(*i*)" and "*i*=*i*+1" until *i* is no longer less than 5.

You can, of course, use any expression at all instead of "*i*<5". In problems #2 and #3, you might test the population vector *n*. For example, to continue to simulate as long as there are still individuals between 0 and 1 years old, you could use "*while* (*n*[0] > 0):". To test whether there are any individuals left at all, you could sum up the entire population and test if it is greater than 0 (and note that there is a nice function *n.sum*() that you have already used).

Problem #1

Modify your code from HW #2 to implement stochasticity and quantization as described above. Start with an initial population array of [750, 450, 360, 240] and a standard deviation of zero. Simulate for 5 generations and check that this population remains constant every generation.

Problem #2

Modify your code from problem #1 to change the standard deviation to sigma=0.05. Use an initial random seed of 0. Instead of simulating for a fixed number of generations, simulate instead until the population dies out completely (i.e., until there are no individuals in any of the four age groups).

Note how many generations are needed for the population to die out completely.

Problem #3

Modify your again to change the standard deviation to sigma=0.2. Otherwise, do exactly the same as in problem #2.

You will note that your code for the three problems is quite similar. We will soon learn to reduce this code duplication by writing your own functions.

Discussion questions:

1. Check your answer for problem #1 manually. Did the computer get the right answers?
2. Did it take longer for your population to die out in problem #2 or problem #3? (If you like, you may repeat problems #2 and #3 using other random seeds, to check if the pattern holds – but you do not need to turn those in). Compare this with problem #1, where you used all of the same parameters except for using sigma=0. Can you explain why a bigger

standard deviation causes populations to die away more quickly? (you do not need a mathematical proof, but just a sentence or two of intuition).

3. If we had not used quantization, then would the populations ever have died out completely?

Extra credit

- We have been using l_x-m_x tables for our population models. Another common way to structure the vital rates is by using *Leslie matrices*. A Leslie matrix holds the same data as L_x-m_x tables, but uses a single sparse matrix. You can find more information on Leslie matrices from Wikipedia, or from Chapter 5 of *Plant and Animal Populations, Methods in Demography* by Thomas Ebert (on reserve at Tisch).
- According to the Perron-Frobenius theorem, only one of the eigenvalues of a Leslie matrix can be positive. The reason this is important is that any eigenvector of a Leslie matrix represents a stable population state, and the magnitude of the eigenvalue tells how robust the population growth is. Why might that be? (It is actually quite intuitive, given the basic definition of what an eigenvector is).

Logistics:

- Use any lab PC system to write your code. You may use your own laptop if you prefer.
- The due date for this assignment is on the class calendar
- Submit your project at https://www.ece.tufts.edu/ee/194MSO/provide.cgi, which is also accessible from the course web page. You should turn in two files: HW3.py and discussion.pdf (or whatever other format you use). The HW3.py file should clearly show which code is for which of the three problems.