Workshop 6: Stochasticity

Everything we have done so far has been deterministic, meaning that if we start out with the same initial values, we get the same output, every time. Even chaotic dynamics in the Logistic Growth model are deterministic. In nature, however, many events are stochastic (e.g., a healthy individual might get stepped on and die, or a storm may kill many individuals). Here, we will learn how to incorporate various stochastic processes into models.

Probability distributions

We have already seen how to draw samples from various distributions using commands such as `runif` (uniform distribution) and `rnorm` (Normal distribution). These are both *continuous* probability distributions. The exponential distribution is another commonly used continuous distribution. *Discrete* probability distributions including the Bernoulli, Binomial, Multinomial, Poisson, Geometric, and Hypergeometric are all useful when incorporating stochasticity into a model. Let's start off by exploring some of these distributions.

Uniform

The uniform distribution is one of the simplest. By default,

```
runif(n=1)
```

will draw a single sample from a uniform distribution over the range $[0, 1]$. You can change the limits by setting the `min` and `max` arguments. E.g.,

```
runif(n=1, min=-2, max=3)
```

will draw a sample from a uniform over the range $[-2, 3]$. See if you can convince yourself that this is actually the same as

```
runif(n=1)*5 - 2
```

Binomial

Suppose I have N individuals, and I want to distribute them among two classes (this could be determining "survivors" and "non-survivors" during a die-off event or assignment of "males" and "females" among offspring). For our purposes, let's suppose we are modelling survival and that the probability of survival is 0.7 for every individual. The Binomial can quickly let us know how many individuals, from some total number, survive. E.g.,

```
rbinom(n=1, size=80, prob=0.7)
```

You can read the above code as "Give me a single draw from a binomial distribution with a size of 80 and a probability of success of 0.7." In our example, this would tell us how many of the 80 individuals survived. If you wanted to generate several draws from this distribution, you could insert the above code into a `for` loop or an `apply` statement. However, the better option is to change `n`, which specifies the number of draws! So, compare the below two lines of code:

```
rbinom(n=10, size=100, prob=0.7)
sapply(1:10, function(x) rbinom(n=1, size=100, prob=0.7))
```

Note that, in the second line, I have defined my own function of `x` within the `sapply` statement, but the argument `x` is not actually used inside the function. This is a quick way to use an `apply` statement to repeat a command many times.

1. Calculate the mean of the output from each of the above commands, but with much larger `n` (e.g., try `1e6`), to confirm that the output is, on average, what you'd expect.

You may have noticed that the `sapply` statement took longer? If not, try again, and pay attention to how long each command takes. This is a great example of where knowing a bit about how R works can help with speed. R is *very* fast at drawing many random numbers from probability distributions, however, it is slow at iterating. Consequently, asking R for $10,000$ draws from a distribution once is *much* faster than asking R for a single draw from a distribution $10,000$ times.

If you want to time your code, here's some useful syntax:

```
time.start <- proc.time()
# insert the code you want to time here
time.end <- proc.time()
time.end-time.start
```

`elapsed` tells you how long it took.

Poisson

In class, we incorporated a Poisson into the Lotka-Volterra competition model. The Poisson is a one parameter distribution with λ specifying both the mean and the variance. To draw a single value from a Poisson,

```
rpois(n=1,lambda=10)
```

1. Confirm that, for a large number of draws from a Poisson, the mean and the variance both equal `lambda`. R's built in variance function, `var`, may be helpful here.

2. In class, I specified that "the sum of k draws from a Poisson with mean λ" is equal to "a single draw from a Poisson with mean $k\lambda$." Verify that this is correct.

Vector arguments

A useful feature of R is that you can pass vectors in as arguments to probability functions. For example, let's go back to our survival example from the Binomial section. Suppose, instead of a single survival probability, we wanted to compare the number of survivors for 10 different survival probabilities $(0.1, 0.2, \ldots, 1)$. We could use the following:

```
my.probs <- 1:10/10
rbinom(n=10, size=100, prob=my.probs)
```

Here, for the first of the `n=10` draws, `rbinom` uses the first probability, `my.probs[1]`. For the second of the `n=10` draws, it uses the second probability, `my.probs[2]`, and so on. If `n` is less than `length(my.probs)=10`, not all of the probabilities will be used and if it is greater, then it will recycle values, starting at the beginning.

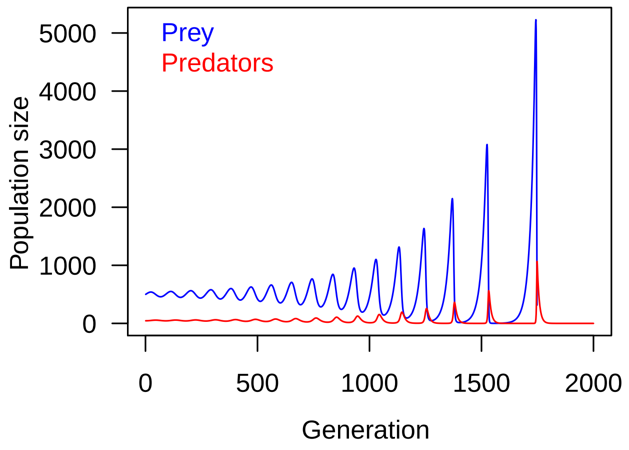
1. Increase the replication to verify that the values output from `rbinom` do, indeed, match those specified by the probability vector. You can do this by simply increasing `n`, but you will need to figure out how to handle the output. Hint: the `matrix` and `rowMeans` commands might be helpful here.

Lotka-Volterra predator-prey model

Here, we'll explore the Lotka–Volterra model of predator-prey dynamics in discrete time. Let $P[t]$ equal the number of predators at time t and $H[t]$ equal the number of prey. The parameters of the model are: the per capita growth of the prey in the absence of the predator (r), the per capita probability that a predator contacts and kills a prey (b), the per capita growth of the predator following the consumption of prey (c), and the death rate of predators (d). With these definitions, the discrete time predator-prey model is

$$H[t+1] = H[t] + rH[t] - bH[t]P[t]$$
$$P[t+1] = P[t] + cH[t]P[t] - dP[t]$$

1. Following the steps from earlier workshops, write code to numerically iterate this model and create a plot showing predator-prey dynamics. For example, for the parameter combination $r = 0.05$, $b = 0.001$, $c = 0.0002$, $d = 0.1$, $H[0] = 500$, $P[0] = 45$, I got:

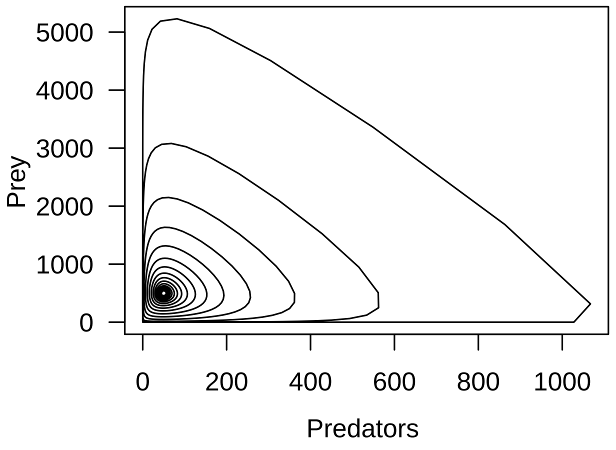


As we can see, the system cycles with increasing amplitude until, eventually, the predators drive the prey extinct, and then they follow suit and also go extinct. To ensure that my population sizes never became negative, I added two `max` commands (one for predators and one for prey) inside my for loop:

```
for(t in 2:num.iter) {
  H[t] <- max(0, [prey equation here] )
  P[t] <- max(0, [pred equation here] )
}
```

This ensures that any negative values are replaced with zero (e.g., extinction).

2. Make another plot, using the same model output, but now plotting the number of predators vs the number of prey, through time. E.g.,

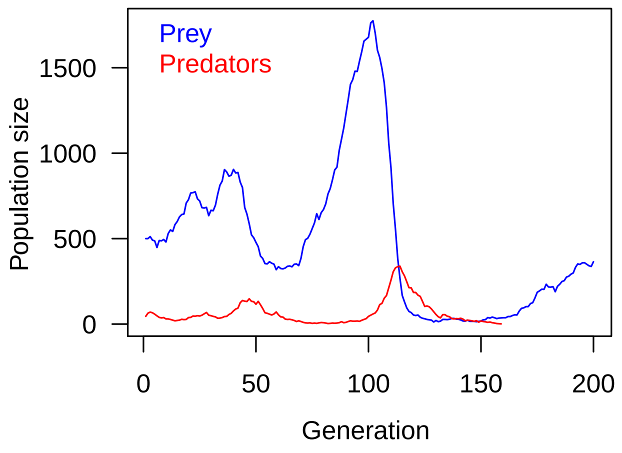


3. If you've stored your output in two vectors, `H` and `P`, see if you can run the following:

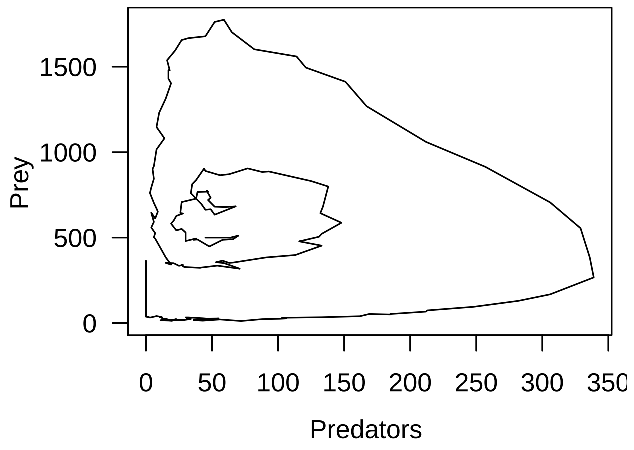
```
plot(NA,
     xlim=c(0,max(P)), ylim=c(0,max(H)),
     xlab='Predators', ylab='Prey', las=1)
for(i in 1:(nrow(out)-1))
  lines(P[i:(i+1)], H[i:(i+1)], col='black')
```

For me, this runs slowly enough that it is effectively an animation!

4. Earlier, we made sure that our numbers of predators/prey were always positive. However, we never checked to ensure that they were whole numbers. Lets go back and do that now, and let's use a Poisson distribution, which is akin to assuming that each predator/prey has a number of offspring that is drawn from a Poisson. Do this, by following the code that we used in class for the Lotka-Volterra competition model. Then recreate each of the above two figures. Here's what mine looked like for the same parameters as above. Yours will look different!



In the above, the predators actually went extinct first and then the prey grew exponentially (I'm only showing the first 200 generations - prey exponential growth happend after that). This contrasts what we had above without stochasticity, where the prey went extinct first, which then led to a gradual extinction of predators. Try recreating your figures a few times to compare iterations.



Setting the seed

It is often useful to be able to recreate the same set of stochastic draws (e.g., in case you want to identify a bug in your code that only happens sometimes). To do this, you can set R's random seed, using the `set.seed` command. E.g., try:

```
set.seed(1) ## set the seed
runif(5)
set.seed(1) ## recreate the above 5 draws using the same seed
runif(5)
set.seed(2) ## try a new seed
runif(5)
```

```
set.seed(2) ## recreate the above 5 draws using the same seed
runif(5)
```