Workshop 3: Numerical Analyses

Now that you are masters of plotting, let's run some models and create some plots. You're welcome to use any plotting commands that you are familiar with (e.g., ggplot, if you are so inclined).

If statements

`if` and `else` (and, in R, you even have `ifelse`) statements are great ways for making decisions based on conditions, or "booleans". For example, try this:

```
for(i in 1:10) {
  if(i<=5) {
    print(paste(i, 'is less than or equal to 5'))
  } else {
    print(paste(i, 'is greater than 5'))
  }
}
```

Note in the above the `i<=5` stands for "i less than or equal to 5." This is a *boolean*. Try replacing the `<=` with each of:

```
<
>
>=
==
!=
```

1. Write a function that takes a single integer as an argument and prints out a statement telling the user whether that integer is odd or even.

Recursive functions

In class, we have learned about discrete time models (aka, "recursion equations"). **Recursive functions** are also common-place in computer programming. A recursive function is a function that calls itself, kind of like looking at a mirror in a mirror.
Now, let's write a basic function that calls itself. We'll use exponential growth as our example:

$$N[t+1] = (1+r)N[t]$$

You might think to implement this with the following code

```
N <- function(r, t) {
  return((1+r) * N(r, t-1))
}
```

But what happens if you try calling this function with, say $r = 0.1$ and $t = 5$?

```
N(r=0.1, t=5)
```

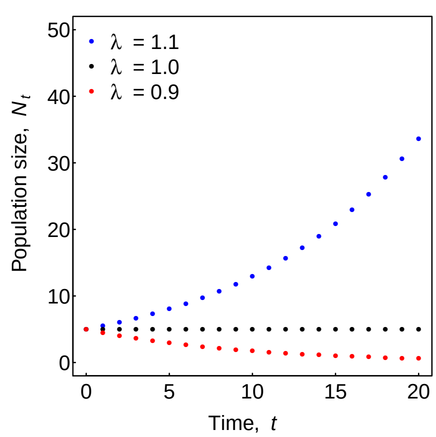
You should get some sort of error. The problem is that, as currently formulated, you've sent the computer into an infinite recursion - the it keeps calling itself... first with $t = 4$, then $t = 3$, and so on. It doesn't know to stop when $t = 0$. So, we need to add a break condition, that gets the recursion to stop, and then backtrack all the way out. We'll use an `if` statement to accomplish this.

```
N <- function(r, N0, t) {
  if(t==0) {
    return(N0)
  } else {
    return((1+r) * N(r, N0, t-1))
  }
}
```

Here, we've also added an argument, `N0` which is our initial population size (e.g., the one at time $t = 0$). Try calling this function with some different values of r, $N0$, and t.

```
N(r=0.1,N0=10,t=10)
```

1. Use your recursive function, and potentially a `for` loop or an `sapply` statement to create a plot similar to the one I showed in lecture 1b (see below).



To add a legend to your plot like I have, you can use the following:

```
legend('topleft',
       legend=c(expression(italic(lambda)~' = 1.1'),
                expression(italic(lambda)~' = 1.0'),
                expression(italic(lambda)~' = 0.9')),
       col=c('blue','black','red'),
       pch=16,
       pt.cex=0.5,
       bty='n')
```

Numerical iteration

While recursive functions are fun, they are not the most efficient way to numerically iterate a model. This is because, in order to calculate `N[5]`, you need to calculate `N[4]`, `N[3]`, and so on. Then, in order to calculate `N[6]`, you need to re-calculate all these again! So, let's do this another way - by just setting an initial population size and iterating forwards in time. First, let's write a function that calculates `N[t+1]` given `N[t]`

```
N.next <- function(r, N.current) {
  return((1+r)*N.current)
}
```

Now, let's suppose we have a population size of 10 and $r = 0.1$ and let's calculate the population size in the next three generations:

```
N1 <- 10
N2 <- N.next(r=0.1, N.current=N1)
N3 <- N.next(r=0.1, N.current=N2)
N4 <- N.next(r=0.1, N.current=N3)
```

and then let's see what we got:

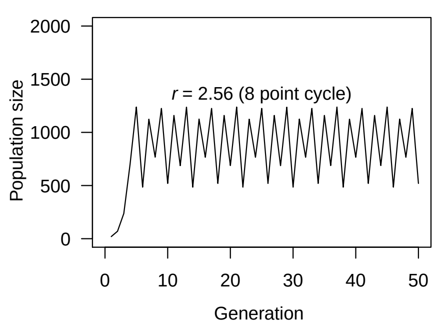
```
c(N1, N2, N3, N4)
```

1. Use a `for` loop to automate this process, calculating the population size over 20 generations.

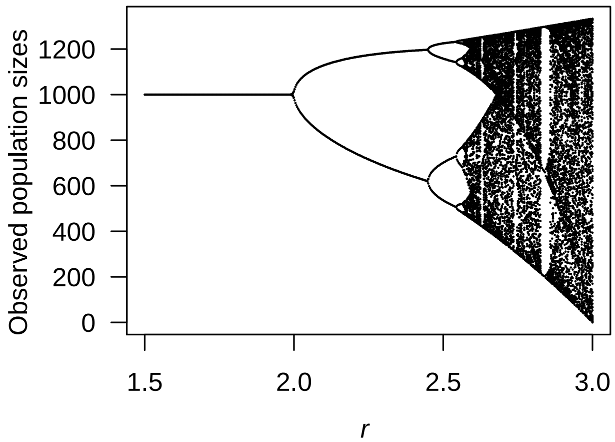
2. The discrete time equation for logistic growth that we examined in class was:

$$N[t+1] = \left(1 + r\left(1 - \frac{N[t]}{K}\right)\right) N[t]$$

   Following the same steps as for the exponential model above, generate a couple plots akin to those I presented in lecture, e.g.,



3. In lecture, I presented a bifurcation plot for the logistic growth model (with $K = 1000$), below



Here, the x-axis is the growth rate, r, and the y axis shows the population sizes at equilibrium. Create a version of this figure yourself. The `seq` command may be helpful for quickly generating a vector of r values to use.