LAB 3 - DISCRETE DYNAMICS INTRO

MATH 1170

4 SEPTEMBER 2018

In this lab, we'll begin to investigate discrete dynamical systems. By the end of the lab, you should be able to

- Define a function in R
- Use R to model a discrete dynamical system
- Implement cobwebbing in R
- Learn about "for" loops

Defining functions

Remember week 1, when we learned how to plot the function $y = e^{\sin(x^2-2)}$? Today, we're going to learn a shortcut to define functions in R using the function command. Instead of just defining an output vector `y`, we will define the actual function which can be used to give us multiple outputs. Let's call this function `CrazyFunction`.

```
> CrazyFunction <- function(x){ exp(sin(x^2-2)) }
```

Let's talk through the pieces of this definition. In the parentheses after the word *function*, we put our input which for now I have called *"x"*. Inside the curly brackets we put the formula for what we are going to do to the input to get output. In other words, the general recipe (*not a command you type in!*) for defining a function is[1]

```
functionName <- function(variableName){ stuff involving
    variable }
```

[1] Note this means that you can call your variable anything you'd like. For instance, this is a perfectly equivalent declaration of the same function using `meow` as the input variable: `CrazyFunction <- function(x){exp(sin(meow^2-2))}`

Why is this useful? Well now the function behaves like a function should! We can, for instance, put in $x = 1$ in the console, we'll get out

```
> CrazyFunction(1)
```

Function definitions make evaluating functions for given values much easier and therefore hopefully make your life less stressful. We can also easily define a set of input and output vectors like we did before.

```
> xvals <- seq(0,10,0.01)
> yvals <- CrazyFunction(xvals)
> plot(xvals, yvals , type="l")
```

We don't even need to bother assigning `yvals` but rather just call our command inside the plot command like so.

```
> plot(xvals, CrazyFunction(xvals), type="l")
```

In general, defining functions is useful because we're likely to use them more than once and R is good at remembering them.

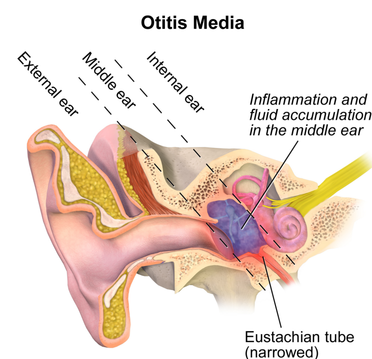
Question 1:

Plot the linear function[2] $y = 2x - 7$ between $x = 0$ and $x = 10$ by defining a function.

[2] or really, any function you'd like. just make it clear which you're plotting

Modeling Multiple Doses of Antibiotic Treatment

Biology background



Physicians treat many patients for otitis media, commonly known as an ear infection. When the middle ear becomes infected with bacteria (such as*Streptococcus pneumonia* or *Pseudomona aeruginosa*) it causes an infection known as otitis media, or more colloquilally, an ear infection. The most commonly prescribed treatment for this type of infection is an antibiotic such as amoxicillin. Amoxicillin functions by inhibiting the synthesis of the cell wall in bacteria which ultimately leads to its demise.[3]

[3] you really don't have to know any of this

Modeling

Let M_t be the amount of Amoxicillin (in mg) in the patient's system at time t. Suppose that the doctor prescribes a patient to take D mg of Amoxicillin every 12 hours for ten days. During a 12 hour period your body metabolizes half the medication that was in your body after the previous dose. We will model this with a discrete time dynamical system.

In this case, the updating function is

$$M_{t+1} = 0.5M_t + D$$

We will now write a new script (using Rstudio) to see how much antibiotic is in one's body after each dose is taken. Be sure to save your script. You can title it anything you would like.[4]

[4] maybe set the mood with `mathisfun.r`, but this might be confusing later

Although you could compute the values manually, we will use our handy function definition instead. First define D and then the updating function, which I have called `update()`.

```
> D <- 250
```

```
> Update <- function(m){.5*m+D}
```

Next, we must start updating the value for each time step. [5]

If the doctor prescribes antibiotic treatment for 7 days, how many times do we need to update the value? Define a list that is the correct length for number of updates we want.

```
> M <- array(0,7)
```

What does this command do? It is a placholder, where it just makes an array (list) of values stored in M that we'll fill in. [6]

We also need a starting value - meaning the amount of antibiotic that is initially in one's body before treatment begins. Define this starting value as `M[1]`. The brackets tell us the value will be placed in the first position of the list M that we just created. [7]

What is a reasonable starting value? Be sure you can justify your choice of this M[1]. Discuss this choice with someone sitting next to you and make sure you agree.

Now, let's update each step, as follows:

```
> M[2] <- Update(M[1])
> M[3] <- Update(M[2])
> M[4] <- Update(M[3])
```

Continue adding more time steps to your script until you have reached the final dose in the treatment.

When you are done updating the function, we need to look at the values you have just computed. We can do this using the print function.

```
> print(M)
```

Now that you have a script written, it is easy to change the dose. Simply redefine D and rerun your code.

Question 2:

What is the final amount of Amoxicillin in the bloodstream after 7 days of treatment if D = 100? 250? 500? Does there seem to be any pattern?

Cobwebbing

Cobwebbing is (usually) a nice way to visualize what is happening when we use an updating function. Each output becomes the input in the next time step which can be visualized as your current y value (output) becoming the next x value (input). This is the same as returning to the line $y = x$ (diagonal) after each time you use the updating function.

There is code provided on my website which will create cobweb plots of the updating function. Please copy and paste it into a new

[5] an important thing to note here is that t isn't actually time, but just indexes when we'll do stuff, $t = 0, t = 1, t = 2, \ldots$. the 12 hours information is a red herring

[6] we initialized all the values in M to 0 but this doesn't matter since we'll override them

[7] an annoying thing worth noting is that many programming languages start with 0 as the first index

script and save it as ''cobweb.R" or just save it somewhere under this game.

Once you have it saved [8], run this script by typing

```
> source("cobweb.r")
```

[8] and find it using the bottom right hand corner, and then set the working directory

Once you have run the script, you will have defined a function called `cobweb()` **that works only for the previously discussed model!**[9] The recipe for calling this function is

[9] if you want to use this for another model, talk to me and I'll coach you through modifying it

```
cobweb(D, steps)
```

That is, the first argument of cobweb specifies the dose, D. The second argument of cobweb specifies the number of time steps. In the console/terminal (not in your script), type the following:

```
> cobweb(250,10)
```

You will see a plot come up where the starting value, $M0$, is shown with a black dot and the ending value, $M10$, is shown with a red dot. The blue points represent the points $(M0, M1)$, $(M1, M2)$, and so on. The green lines show the cobwebbing. In the console the values of $M0, M1, M2, \ldots, M10$ are also printed.

Question 3:

Create three cobwebbing plots (one each for D=100, 250 and 500) to illustrate each of the scenarios from Question 2.

Modeling Fish Populations

For Loops

Now, we'll consider a new model: a newt population.

Every week, you measure that the newts in your yard have reproduced, so that there are 2% more newts added to the current population. [10]

[10] they have better things to do!

Despite being cute, newts are not very smart. As a consequence, a number of newts accidentally wander into the road and are extinguished by the passing-by cars. For the sake of science, you measure this to be roughly $H = 50$ newts per week. Let N_t be the number of newts in the population in a given week. We write the model as follows: [11]

[11] discuss with your neighbor (or self) why this makes sense

$$N_{t+1} = 1.02N_t - 50$$

Suppose we measure that there are initially 1000 newts in the population. We want to know how many newts will be in the population in a year - in other words, in 52 weeks. Let's write a script that can calculate this for us. Again, we initialize an empty array and assign our necessary variables

```
> N<-array(0,52)
> H<-50
> N[1]<-1000

>Update = function(n){1.02*n-H}

> N[2] <- Update(N[1])
> N[3] <- Update(N[2])
> N[4] <- Update(N[3])
...
```

I didn't want to write this out for all 52 weeks. You don't don't want to do this either.[12] Programming is often about being creatively lazy. Here, we can be lazy by using a new tool: a **for loop**. Note that there is a natural "index" for where we are in the process. That is, at the 2nd step, we're doing `N[2]<-Update(N[1])` and generally at the ith step, we're doing `N[i] <- Update(N[i-1])`. Translating this into programming, we get a for loop

[12] I hope

```
> for(i in seq(2,52)) N[i]<-Update(N[i-1])
```

The above line means "for each i from 2 to 52 let N[i] be update(N[i-1])". This is exactly what we want!

Question 4:

How many newts are in the population after one year? Does this number make sense? Why or why not? How long will it take the newt population to go extinct?

Question 5:

In your disgust at the decline of newt populations, you create a safe crosswalk for them to skitter across the road, reducing the number of newts run over per week to $H = 20$. With this change, how many newt are in the population after one year?