DATA 420 Notes – Feb 13-15, 2024

Logistic growth

Let's return to the Aliens vs. Vampires competition. Recall that the explosive nature of exponential growth caused the vampires, once they hit their stride, to easily take over the world. Their numbers just grew and grew and grew without bound...

There's something obviously wrong about this, though: there aren't an unlimited number of humans to bite! Eventually the vampires will exhaust the human population. And this is actually true of pretty much *every* exponential growth situation: since everything in our universe is finite, every exponential growth process will eventually run out of resources.

(Pause to deeply consider this fact and its implications.)

Now how can we incorporate this into our model? Previously, we had V' as:

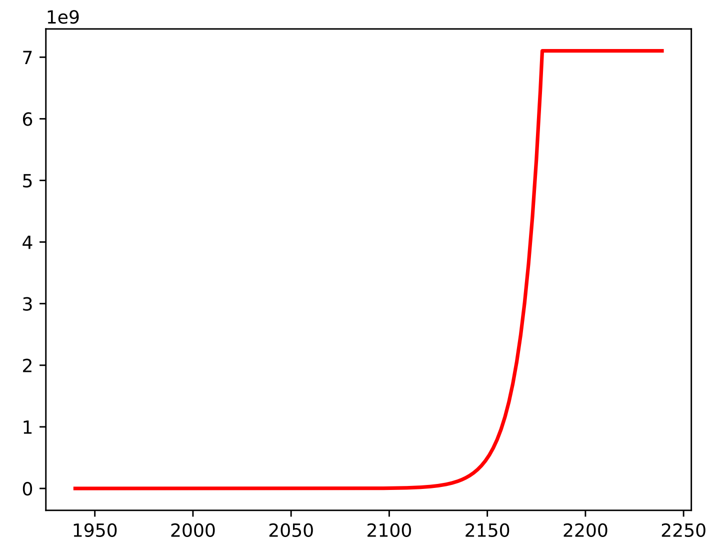
$$V' = bV$$

where the "bloodthirstiness" b was measured in $\frac{\frac{\text{vampires}}{\text{year}}}{\text{vampire}}$. (We rationalized this as "for every vampire now in existence, how many new vampires per year will come into being?")

Suppose we define a parameter P as the total **p**opulation of the earth, which includes both humans and vampires. One first cut at modeling the resource exhaustion would be this:

$$V' = \begin{cases} bV, & V < P \\ 0, & V = P \end{cases}$$

This says: "as long as there are humans left to harvest (*i.e.*, the number of vampires is less than the total population), allow the vampires to grow at our previous rate. As soon as they exhaust the population, stop." It gives us a hard cutoff like this:



Maybe that's how things would work. But probably not. If you think about it, the lower the percentage of the population is human, probably the harder it will be for the vampires to find those few remaining potential victims. So we'd expect that as the population gets vampirized, the vampirization rate would *gracefully* level off to zero, rather than getting suddenly clipped to zero once the last human was extinct.

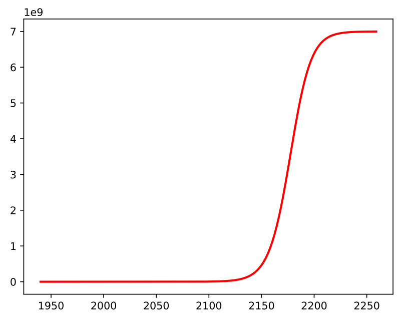
This approach is called **logistic growth**. It's based on the notion of a **carrying capacity**, which in general is the maximum number of individuals an environment can support. A field of dandelions can only support so many dandelions, and the market of consumers for a product can only support so many purchases before it's saturated. In general, some resource places limitation on the growth, and the carrying capacity is the maximum level a population can grow to before it exceeds that resource.

In the present case, the limiting factor is the number of potential victims. The logistic growth factor will be *the fraction of the population which consists of potential victims*. Mathematically, this is $\frac{P-V}{P}$, where P is the total population (including both humans and vampires). Looked at another way, it's the ratio of the number of humans to the total population. Our logistic growth equation will be:

$$\begin{aligned} V' &= bV \cdot \frac{P-V}{P} \\ &= bV \cdot (1 - \frac{V}{P}) \end{aligned}$$

Stare at this and see the logic. We still have bV in there, but now we're multiplying it by a factor that will smoothly go from 1 (when everyone is human) to 0 (when nobody is). This gradually reduces the rate of vampirization until the victims are no more.

Graphically, we get what's known as a **sigmoid** curve:



Here's the general form this differential equation is often written in:

$$X' = rX \cdot (1 - \frac{X}{K})$$

where X is any logistically-growing population of things, r is the growth (or "reproductive" factor (in $\frac{\text{things/time}}{\text{thing}}$), and K is the carrying capacity (in things). (Double-check for yourself that the units work out.)

This equation applies to populations, economies, diseases, social phenomena, and a whole host of other things. It was discovered in the 19th century and is without a doubt one of the most important equations of all time.

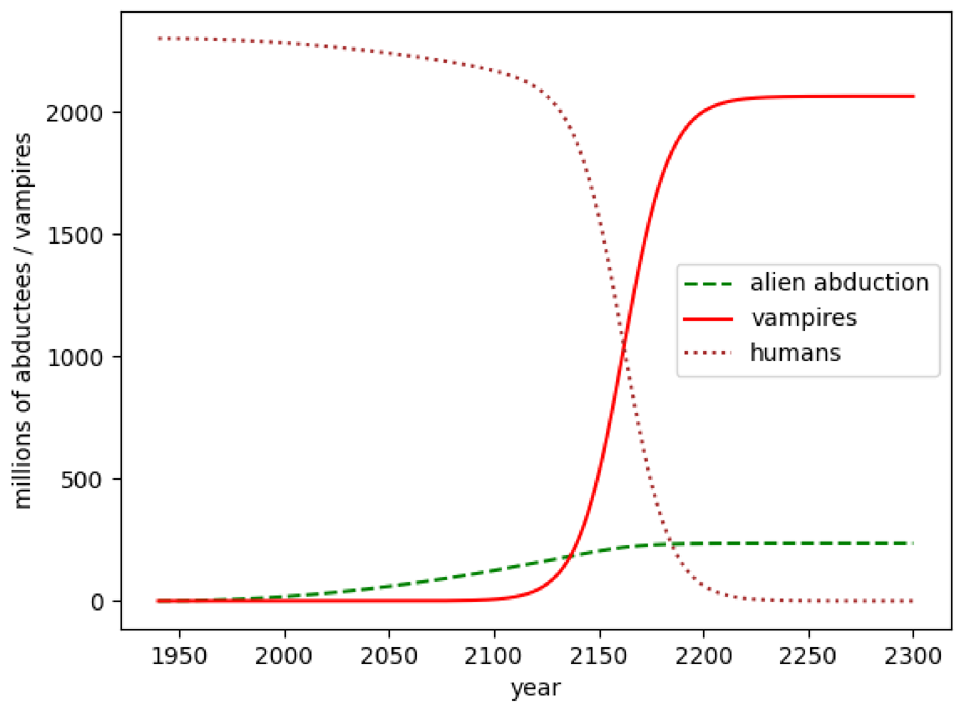
Implementing this in Python simply involves incorporating the `logistic_factor` into the code, and multiplying each prime by it. Look carefully at the first line of the `for` loop's body. We're calculating `logistic_factor` for the current iteration as:

$$1 - \frac{V_{i-1} + A_{i-1}}{V_{i-1} + A_{i-1} + H_{i-1}}$$

(H stands for "humans.") Do you see the logic? We're looking at the previous time step ($i - 1$) and asking, "what fraction of the population is still available to be abducted/bitten?" When A_{i-1} and V_{i-1} are both still low (near the beginning of the simulation) the entire quantity, above, is about 1. When A_{i-1} and V_{i-1} grow high enough to rival and overtake the remaining human population, the entire quantitive, above, shrinks to around 0. That's why we get the desired effect when we multiply the `prime` values by this logistic factor.

```
delta_x = 1/365    # years
start_x = 1940     # year
end_x = 2122       # year

init_pop = 2.3e9   # earth's (human) population, circa 1940

# Given an array index, return the corresponding time (year).
def itox(i):
    return delta_x * i + start_x

# Given a point in time (year) return the corresponding array index (unitless).
def xtoi(x):
    return int((x - start_x) / delta_x)

# How aggressive are the aliens? At what rate do they abduct victims for every
# year after 1940?
aggressiveness = 10000   # (abd/year)/year

# How thirsty are the vampires? How many does each one bite, on average, in a
# year?
bloodthirstiness = .1   # (vampires/year)/vampire

# Our x-values: the precise times at which we will measure and compute the
# quantities of interest.
x = np.arange(start_x, 2300, delta_x)   # years

# Our "stock variables": one for the number of UFO-abducted victims, one for
# the number of vampires, and one for the number of potential victims still
# available to be snarfed. They are arrays, of course, because we want to track
# how many of each quantity there are *at each time period*.
A = np.zeros(len(x))   # abductions
V = np.zeros(len(x))   # vampires
H = np.zeros(len(x))   # humans

# Set our initial conditions. When the simulation begins at the stroke of
# midnight New Year's Day 1940, there's nobody on spaceships yet, and there's
# one lonely vampire in the world.
A[0] = 0
V[0] = 1
H[0] = init_pop

# The main simulation loop. For each time period...
for i in range(1, len(x)):

    logistic_factor = 1 - (V[i-1] + A[i-1]) / (V[i-1] + A[i-1] + H[i-1])

    # ...calculate the rate of each quantity at this point in time, and...
    Aprime = (itox(i) - start_x) * aggressiveness   # abd/year
    Aprime *= logistic_factor
    Vprime = V[i-1] * bloodthirstiness              # vamp/year
    Vprime *= logistic_factor

    # ...increment the quantities by that amount, times our time period.
    A[i] = A[i-1] + Aprime * delta_x                # abd
    V[i] = V[i-1] + Vprime * delta_x                # vamp
    H[i] = H[i-1] - Aprime * delta_x - Vprime * delta_x

plt.plot(x, A/1e6, color="green", linestyle="dashed", label="alien abduction")
plt.plot(x, V/1e6, color="red", label="vampires")
plt.plot(x, H/1e6, color="brown", linestyle="dotted", label="humans")
plt.xlabel("year")
plt.ylabel("millions of abductees / vampires")
plt.legend()
plt.show()
```

The output looks like this when it's run:



The vampires overtake the aliens, as before, but it's no longer quite as massive a whipping – since there are only so many humans to consume, the aliens do end up with a modest piece of the pie.

The Lotka-Volterra model (predator-prey)

This central insight of logistic growth is used in a slightly different way in the hugely influential Lotka-Volterra predator-prey model, invented independently by Alfred Lotka and by Vito Volterra in the early 20th century.

The idea is as follows. Suppose we have two species (say, **b**ats and **m**ice) where one preys on the other for food. Let's say the number of bats at time x is B_x and the number of mice is M_x. Then, assuming the populations are well-mixed[*] the number of *possible encounters* between them is proportional to the *product* of B_x and M_x. In symbols:

$$\#\text{ possible encounters}_x \propto B_x \times M_x$$

(All the "proportional to" symbol $\propto$ does is let us state this fact without having to specify the value of the constant (called the **constant of proportionality**).

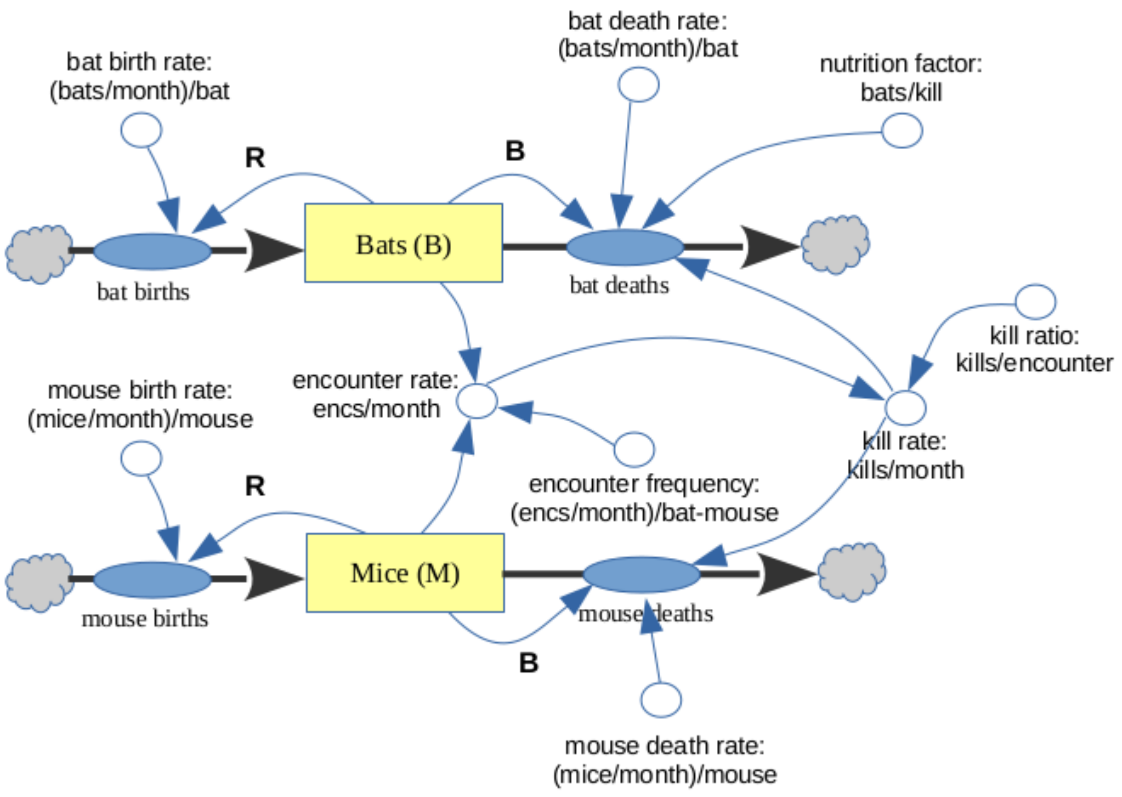
[*]Meaning, they all have equal access to each other in a sort of well-stirred soup.

This makes sense when you think about it. There are $B_x \cdot M_x$ different possible pairs of individual bats-and-mice. If each pair has an equal opportunity of "happening" within a given time period, then the number of possible encounters is directly related to this product.

Let's call the constant of proportionality the **encounter_frequency**, which is measured in units of $\frac{\frac{\text{encounters/time}}{\text{bat}}}{\text{mouse}}$. ("For every mouse we have...then for every bat we have, how many encounters do we expect per unit time?") Let's call the proportion of these encounters which end successfully for the bat (and in death for the mouse) the **kill_ratio**. Now we have:

$$\left.\frac{\#\text{ kills}}{\text{time}}\right._{x} = B_x \times M_x \times \text{encounter\_frequency} \times \text{kill\_ratio}$$

This dynamic is the heart of the Lotka-Volterra model.


bat death rate: (bats/month)/bat
bat birth rate: (bats/month)/bat
nutrition factor: bats/kill
R
B
Bats (B)
bat births
bat deaths
kill ratio: kills/encounter
encounter rate: encs/month
mouse birth rate: (mice/month)/mouse
kill rate: kills/month
R
encounter frequency: (encs/month)/bat-mouse
Mice (M)
mouse births
mouse deaths
B
mouse death rate: (mice/month)/mouse

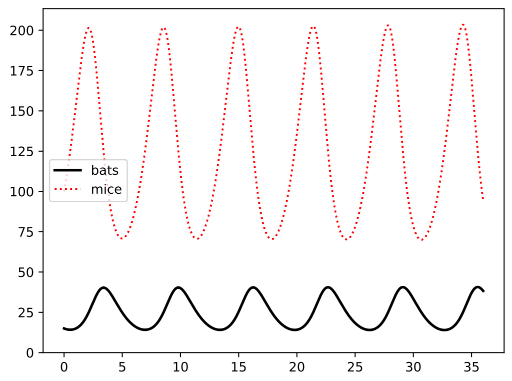
A complete model

Converting into Python, we get:

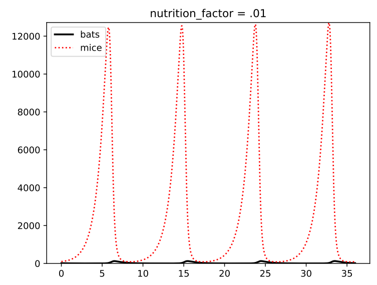
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

delta_x = 1/30                  # mo
x = np.arange(0,240,delta_x)    # mo

batBirthRate = 1.2              # (bats/mo)/bat
mouseBirthRate = 1.2            # (mice/mo)/mouse
batDeathRate = 2.1              # (bats/mo)/bat
mouseDeathRate = 1.1            # (mice/mo)/mouse
nutritionFactor = 2             # bats/kill
killRatio = .05                 # kills/encounter
encounterFreq = .02             # encs/mo/bat/mouse

B = np.empty(len(x))
M = np.empty(len(x))
B[0] = 10
M[0] = 30

for i in range(1,len(x)):
   encounterRate = encounterFreq * M[i-1] * B[i-1]   # encounters/mo
   killRate = killRatio * encounterRate              # kills/mo
   batBirths = batBirthRate * B[i-1]                 # bats/mo
   batDeaths = batDeathRate * B[i-1] - killRate * nutritionFactor # bats/mo
   mouseBirths = mouseBirthRate * M[i-1]             # mice/mo
   mouseDeaths = mouseDeathRate * M[i-1] + killRate  # mice/mo

   Bprime = batBirths - batDeaths
   Mprime = mouseBirths - mouseDeaths

   B[i] = B[i-1] + Bprime * delta_x
   M[i] = M[i-1] + Mprime * delta_x

plt.plot(x,B,color="black",label="bats",linewidth=2,linestyle="solid")
plt.plot(x,M,color="red",label="mice",linewidth=1,linestyle="dotted")
plt.ylim(bottom=0)
plt.legend()
plt.show()
```

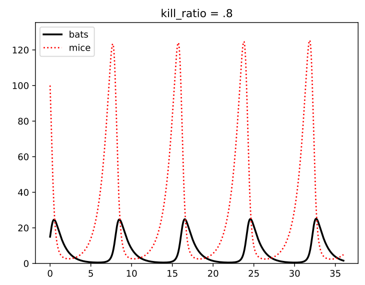
Now close your eyes and try to visualize what this simulation will produce. Will we get an exponential growth of both populations? Or only of one? Or neither? Or will it approach an equilibrium?

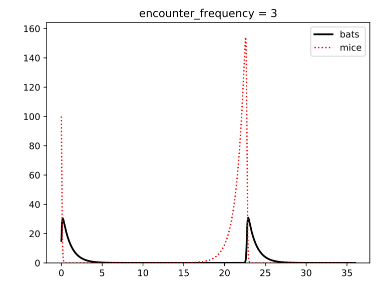
If you're like I was, you'll be dumbfounded to see the following oscillatory behavior:

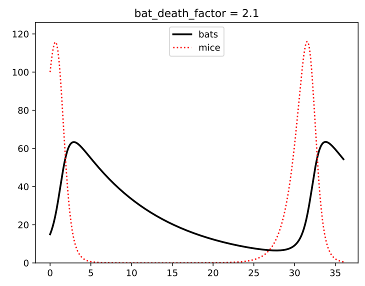


The populations swing wildly back and forth, sometimes over order-of-magnitude ranges. Suppose we make the mice *less* nutritious, or the bats *better* hunters, or the mice *easier* to find, or the bats less likely to die of other causes:









(In case you're concerned, we dropped to a low of .0109 mice in that last run.)

If these results all line up with your intuition, pat yourself on the back. They certainly don't with mine!