DATA 420 Notes – Feb. 3, 2026

Balancing and Reinforcing loops

Feedback loops can be balancing or reinforcing. The difference between the two is: *if the stock's going up tends to keep it going up, it's reinforcing. If the stock's going up tends to prevent it from going up further, it's balancing.*

There's actually two different ways to achieve each of them:

1. If the increase in a stock tends to *increase* an *inflow* to that stock, it's reinforcing.
2. If the increase in a stock tends to *decrease* an *inflow* to that stock, it's balancing.
3. If the increase in a stock tends to *increase* an *outflow* to that stock, it's balancing.
4. If the increase in a stock tends to *decrease* an *outflow* to that stock, it's reinforcing.

Examples:

1. Example: Population growth. When there are more people, there are more babies, and hence an increase in the inflow to people.
2. Example: Resource capacity. The more people walk on to the dance floor, the more crowded the dance floor, which means fewer people will want to walk on to the dance floor.
3. Example: Substance decay. The more of a substance exists, the more of it will cease to exist.
4. Example: The more money you have, the better you're able to avoid paying fees and taxes, which means the more money you keep.

Drug model revisited: dosage schedule

Our previous drug example had the patient taking only a single dose at the beginning of (simulation) time. What if we wanted to monitor the drug's concentration in their plasma as they took doses at periodic intervals?

Let's create a couple more variables, to represent the size of each `dose` (in micrograms) and the `dose_interval` of time that should elapse between each time the patient takes them. When we initialize our `D` vector, we'll set it equal to the first dose:

```
dose = 2 * 325 * 1000                          # ug  ("u" = "micro")
dose_interval = 8                              # hrs

D = np.empty(len(x))                           # ug
D[0] = dose
```

This is a dose of two 325 mg tablets every eight hours. In order to get the patient to take the additional doses, we'll check the clock time at each simulation interval. Only if the clock time is a whole multiple of the `dose_interval` will we add the new dose. This will require an `itox()` function, to convert simulation clock tick (unitless) to wall clock time (hours):

```
def itox(i):
    return delta_x * i + 0
```

(Adding 0 may seem silly, but I do it for consistency – our simulation won't always start at 0. For instance, aliens vs. vampires started at 1940, you'll recall.)

Now, at the bottom of the loop, we'll add the next dose only at the correct times. First let me write it in an intuitive, but technically wrong, way:

```
    if itox(i) % dose_interval == 0:   # technically wrong
        D[i] += dose
```

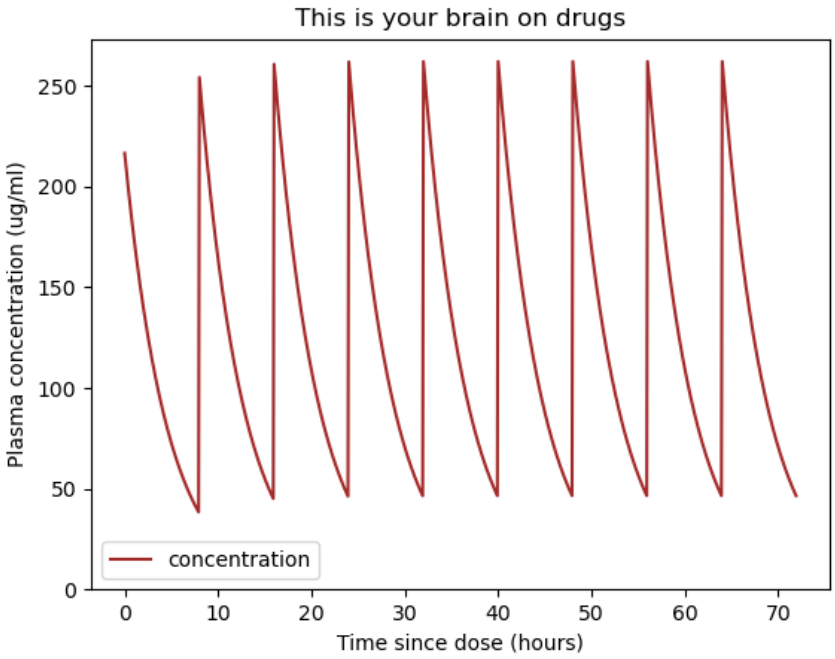
This should make sense to you if you remember the meaning of Python's "`%`" (pronounced "modulo" or "mod") operator. It gives *the remainder after dividing its two operands.* So for instance, if `a`=7.25 and `b`=2, then `a%b` equals 1.25. (After removing all whole "twos" from the number 7.25, you're left with 1.25 at the end.) And if `a`=8 and `b`=2, then `a%b` equals 0. This is what we want: increase the amount of drug in the plasma – *i.e.*, pop your pills – only when the clock time is 8, 16, 24, and so forth.

Now the reason it's technically wrong is that it's not a good idea to compare floating-point numbers for equality in a computer program. The reason gets a bit esoteric, but it's basically because teensy inaccuracies and round-off errors can cause two floats, even though they ought to be equal, to be juuuuuust a litle bit different. So the safe thing to do is change the above code to:

```
    if math.isclose(itox(i) % dose_interval, 0):   # correct
        D[i] += dose
```

The `math.isclose()` function takes two arguments and returns `True` if they are "really really close to each other, or actually the same." File this away in your mind, for this is the way you should always compare floating-point numbers for equality.

If we run our modified program for three simulated days, we get this plot:



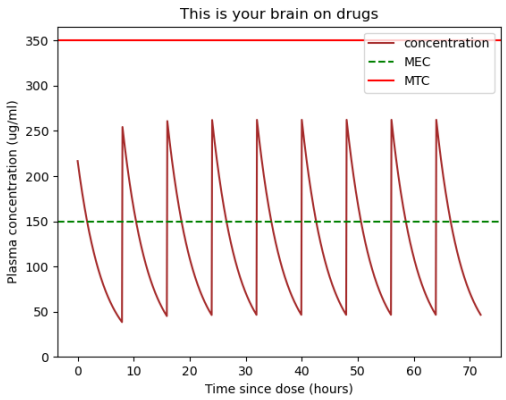
Wow, this patient's drug concentration is really bouncing rail to rail! If you study the plot, you'll realize the problem: by the time the patient takes their next dose, their drug concentration from the *previous* dose will long since have worn off from the previous dose, which means their troublesome symptoms will have to be endured in the interim.

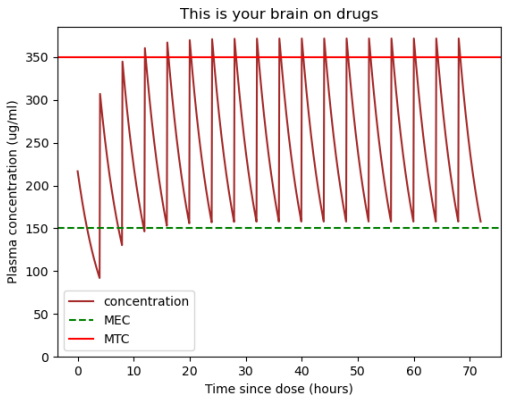
Speaking of symptoms, here are a couple important pharmacological numbers for any drug: the **MEC** ("Minimum Effective Concentration") and the **MTC** ("Maximum Therapeutic Concentration"). These represent the minimum and maximum concentrations that we want the patient's bloodstream to have at all times. If their drug concentration is lower than the MEC, it won't help. If it's higher than the MTC, they could experience dangerous side effects.

Let's add a line for each of these to the plot. We'll use numbers for aspirin, which Wikipedia tells me has an MEC of 150 μg/ml and an MTC of 300:

```
plt.axhline(y=150, color="green", linestyle="dashed", label="MEC")
plt.axhline(y=350, color="red", linestyle="solid", label="MTC")
```

This confirms that the patient doesn't always have enough aspirin in their bloodstream to help their arthritis. (See left side of figure below.) However, if we bump up the frequency to one dose every *four* hours, we get the right side of the figure, which is darn close to what we want:





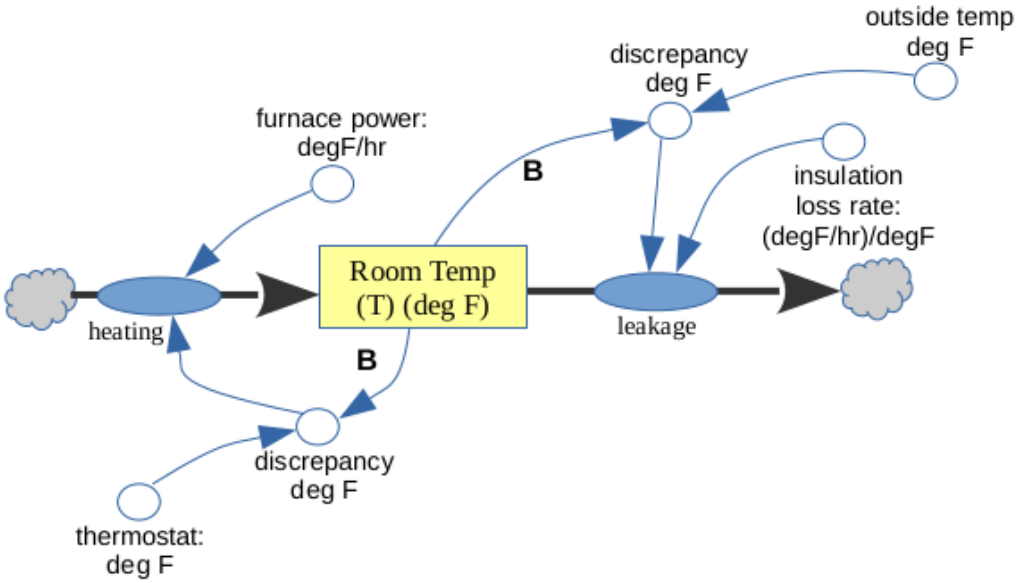
Playing around with the dosage amount and interval should enable you to design a dosage schedule for a patient with a given plasma volume. Keep in mind that not everyone has exactly 3 liters of plasma, so you need to have some room for error! Also, some medications are designed for an "extended release" which essentially increases the half-life, and these can help the patient's drug concentration at a manageable level.

For reference, here is the entire program:

```
import numpy as np
import matplotlib.pyplot as plt
import math

simulation_hours = 24*3                          # hrs
delta_x = 5/60                                   # hrs
x = np.arange(0,simulation_hours,delta_x)        # hrs

half_life = 3.2                                  # hrs
plasma_volume = 3000                             # ml
elimination_constant = math.log(2)/half_life     # 1/hr

dose = 2 * 325 * 1000                            # ug  ("u" = "micro")
dose_interval = 4                                # hrs

D = np.empty(len(x))                             # ug
D[0] = dose                                      # ug

def itox(i):
    return delta_x * i + 0

for i in range(1,len(x)):
    elimination = elimination_constant * D[i-1]   # ug/hr
    Dprime = -elimination                         # ug/hr
    D[i] = D[i-1] + Dprime * delta_x              # ug
    if math.isclose(itox(i) % dose_interval, 0):
        D[i] += dose                              # ug

plasma_concentration = D / plasma_volume          # ug/ml

plt.plot(x, plasma_concentration, color="brown", label="concentration")
plt.xlabel("Time since dose (hours)")
plt.ylabel("Plasma concentration (ug/ml)")
plt.title("This is your brain on drugs")
plt.axhline(y=150, color="green", linestyle="dashed", label="MEC")
plt.axhline(y=350, color="red", linestyle="solid", label="MTC")
plt.ylim(bottom=0)
plt.legend()
plt.show()
```

Two competing balancing loops – thermostat

For our next model, we'll look at a common Systems Dynamics archetype: the pattern with two competing balancing loops. One example of this pattern is a house with a heater (or A/C) and a thermostat:



There are two balancing loops, one of which is "trying" to get the T stock to be equal to the thermostat setting, and the other of which is "trying" to get it to be the outside temperature.

Even though things appear symmetrical, however, these loops will work somewhat differently. The left loop will either be turning the furnace on or off, depending on whether the thermostat is higher or lower than the current temperature. The "furnace rate" parameter controls how many degrees per hour the furnace is able to heat the house when it's on full blast.

The right loop, on the other hand, will be *constantly* degrading the T stock, and by greater amounts the farther apart the house and outside temperatures are. The "house leakage factor" represents the quality of the insulation. Note carefully its units: $\frac{\frac{degF}{hr}}{degF}$. Put another way, "for every degree by which the inside and outside temperature differ, how many degrees per hour will the house cool on its own?"

Here's a complete implementation:

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

thermostat = 68               # degF
outside_temp = 40             # degF
insulation_loss_rate = .2     # (degF/hr)/degF
furnace_power = 4             # degF/hr

delta_x = 1/60                # hrs
x = np.arange(0,24,delta_x)   # hrs

T = np.empty(len(x))          # degF
T[0] = 67                     # degF

furnace_on = np.empty(len(x), dtype=bool)
furnace_on[0] = True

for i in range(1, len(x)):
    if not furnace_on[i-1] and T[i-1] < thermostat - 1:
        furnace_on[i] = True
    elif furnace_on[i-1] and  T[i-1] > thermostat + 1:
        furnace_on[i] = False
    else:
        furnace_on[i] = furnace_on[i-1]

    if furnace_on[i]:
        heating = furnace_power   # degF/hr
    else:
        heating = 0

    leakage = insulation_loss_rate * (T[i-1] - outside_temp)  # degF/hr

    Tprime = heating - leakage    # degF/hr
    T[i] = T[i-1] + Tprime * delta_x


plt.plot(x,T,color="orange",label="inside temp")
plt.plot(x,furnace_on * 10,color="gray",linewidth=2,label="furnace on/off")
plt.axhline(y=outside_temp, color="blue", linestyle="dashed",
    label="outside temp")
plt.axhline(y=thermostat, color="brown", linestyle="dotted", label="thermostat")
plt.xlabel("hours")
plt.ylabel("deg F")
plt.ylim(bottom=0, top=100)
plt.legend()
plt.show()

print(f"The furnace was on {furnace_on.sum()/len(furnace_on)*100:.1f}% today.")
```

Some items of note:

1. We're keeping track of an array called "`furnace_on`", which will be a boolean indicating whether the furnace was on or off at each simulated time click (in our case, each minute). We do this so we can analyze it and plot it at the end.

2. You might have expected the first if-statement to look different. Why wouldn't it simply be this, you might ask:

```
if T[i-1] < thermostat:
    furnace_on[i] = True
else:
    furnace_on[i] = False
```

   At first blush, this looks like exactly the right thing: if the current temperature is lower than the thermostat, turn the heat on; otherwise, turn it off.

   The problem is that this is going to lead to quickly oscillating behavior in which the furnace turns on and off many times a minute (or even per second!) Suppose we have the thermostat set for 68, and it's currently 67.99 degrees. We turn on the heat, and in a few seconds, it's 68.01 degrees. So we turn it off, and in a few seconds, it's 67.99 degrees. So we turn it on...and turn it off...and turn it off...

   Such behavior would be annoying in a real home (besides wearing out the furnace through constant startup and shutdown). In real life, we want to incorporate **hysteresis**, which means that we'll establish a buffer zone of sorts that will prevent constant startups and shutdowns. The idea is this:

   > "If our target temperature is 68, let's keep the heat on until it's actually *69* (or so). Then, when we turn the heat off, let's keep it off until it drops all the way to 67 (or so)."

   There's nothing magic about a buffer zone of ± 1 degree*; we could do ± 2 degrees, or $\pm .5$ degrees, or anything else. The tradeoff is clear: the larger our hysteresis buffer, the less precisely we will keep the home at exactly the desired temperature, but the less frequently the furnace will toggle.

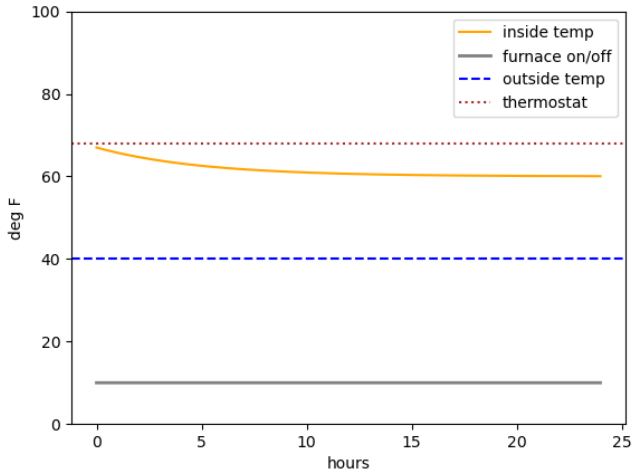
3. Note that the heating flow is effectively binary – the furnace is either on (at full blast) or off. The leakage flow, however, is smoothly varying, based on the discrepancy between `T[i-1]` and `outside_temp`.

4. You might wonder what happens when we "plot" the `furnace_on` variable, given that it's an array of booleans. The answer is that Python treats each `False` value as a 0, and each `True` value as a 1. That's not quite enough difference for us to see on the plot (since our y-axis will range over 50 degrees or so), so we'll just multiply the vector by something (here I chose 10) to make its variations more apparent.

*The symbol "$\pm$" means "plus or minus."

5. Finally, we print out how much of the day the furnace was running by simply taking the "average" of the `furnace_on` array.

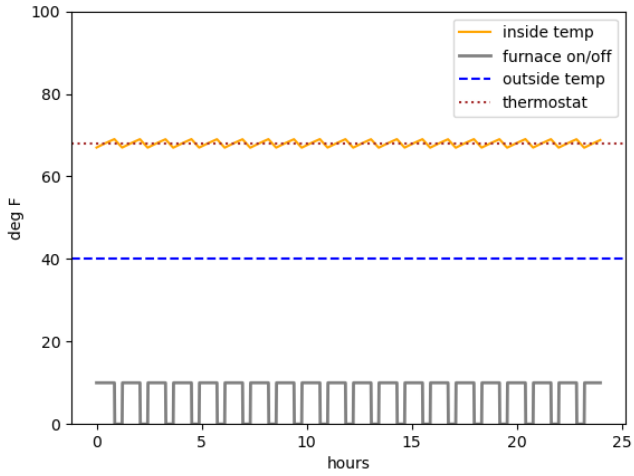
When we run the simulation with the above numbers, we get this plot:



and this output:

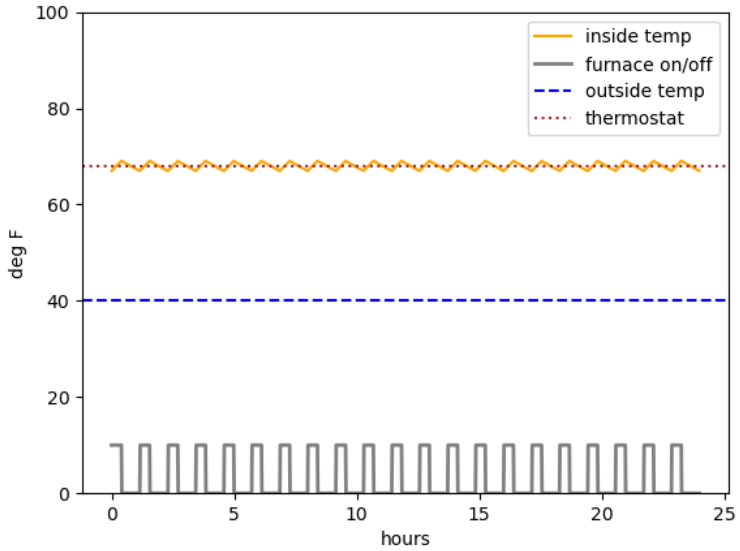
```
The furnace was on 100.0% today.
```

Yikes! Turns out our wimpy furnace isn't able to keep the house warm, even running full blast all day. :(Let's jack up the `furnace_power` from 4 to 8:




```
The furnace was on 71.0% today.
```

The house is now comfortable, but man, 71% utilization still seems like a lot. Perhaps if we improved the insulation. Let's change `insulation_loss_rate` from .2 to .1 and give it a go:




```
The furnace was on 35.1% today.
```

Now that's more like it. The orange jaggedy line doesn't look very different to the eye, but you can see from the gray pulse at the bottom that the furnace is on for a much smaller time period each time it cycles.

There's lots of other interesting ways this simulation could be enhanced. In the real world, of course, the outside temperature isn't totally constant (it varies both with time of day and day of the year), and neither is the thermostat (many people set it lower when they go to work and when the sleep, only kicking it up a notch when they're active and awake at home.) That's the great thing about modeling and simulation: you can experiment with changes to the model and the environment easily, without having to actually do anything to the (stubborn, costly, ethically dubious) real world.