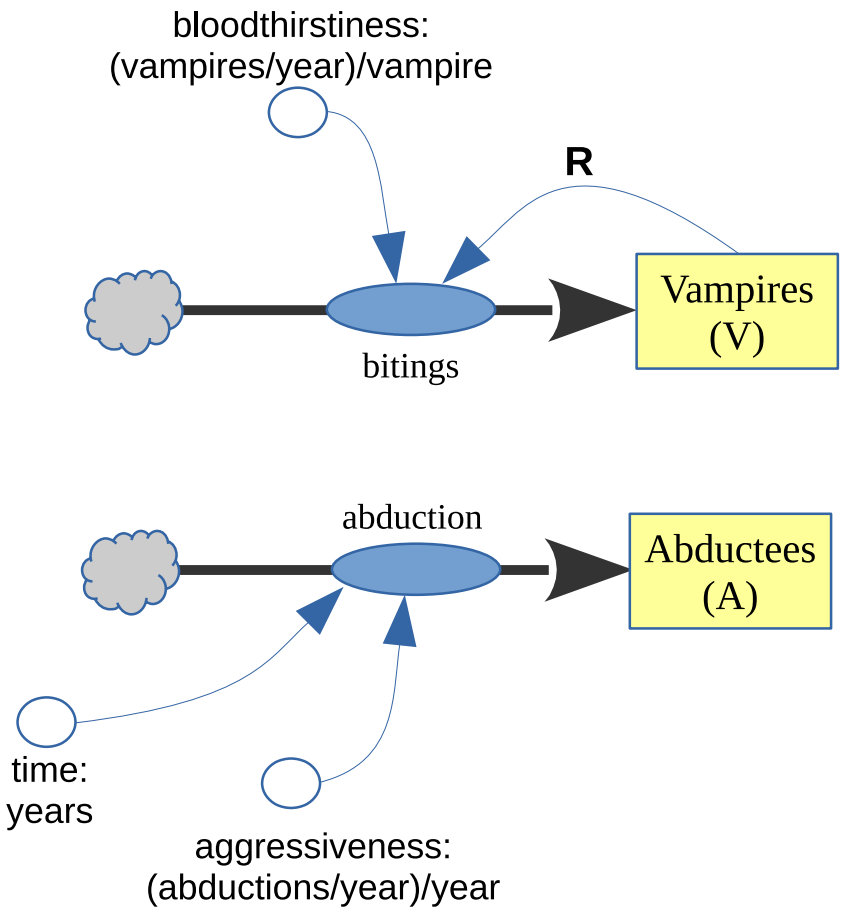
DATA 420 Notes – Jan. 29, 2026

Simulating System Dynamics (SD) models

A System Dynamics problem presents itself as a complex interaction of different variables over time. Our goal in understanding such a system is to formulate a model of it, and then simulate the model. The results of the simulation will give us a good idea about how the complex system will act over time when it is given certain initial conditions. Even simple looking models are often not possible to solve analytically, and hence the simulation tells us something that is not feasible to determine any other way.

Stock-and-flow diagrams



Our main task is to turn a System Dynamics diagram (like the one above) into a running program.

First, some basic terminology and concepts about these so-called "stock and flow" diagrams:

stock variable. These are represented on the diagram by boxes. A stock variable is typically something that accumulates as time goes on: it is a quantity that goes up or down in value, and we want to track how it changes over time. For this reason, we will normally use an array/vector to represent it, so that its value at each moment in time can be preserved.

derived stock variable. These are represented on the diagram by any white circle that has an incoming arrow from a box (or from another derived stock variable.) If a white circle has an incoming arrow from a box, then it is essentially another stock variable: we are interested in tracking how it accumulates or decreases over time. The only way in which it differs from a box is that its value is usually based on a simple calculation *from* a box. Hence we can give our main attention to tracking the box's value as it changes throughout the simulation, and then make a simple conversion of those values to obtain the derived stock variable's values. For the same reason as above, a derived stock variable will normally be an array of values. (By the way, there aren't any derived stock variables in the aliens/vampires example, above. There is one in the example later in these notes, however.)

variable. These are represented on the diagram by white circles. Each one will correspond to a variable in the program. These usually have constant (unchanging) values, and so are represented in our simulation programs as scalars (not arrays). "Bloodthirstiness" and "aggressiveness" are two examples.

cloud. "Clouds" in the diagram represent boundaries on our area of interest. As System Dynamics pioneer Dana Meadows says, every system interacts with an environment, and it's up to us where exactly we want to draw the system boundary. Whenever we have a cloud, it effectively means "comes from somewhere, in unlimited quantities" or "goes to somewhere in unlimited quantities" and that's that.

flow. Finally, a flow is depicted on the diagram by an oval on top of a thick, arrowed line. It represents a quantity whose value will change over time, and which will influence the value of some stock variable. Though it is a scalar, it is a dynamic (changing) quantity, and hence will be continually computed in the main loop of the simulation in order to help properly calculate the values of the stock variable(s) it influences.

Outline of basic procedure

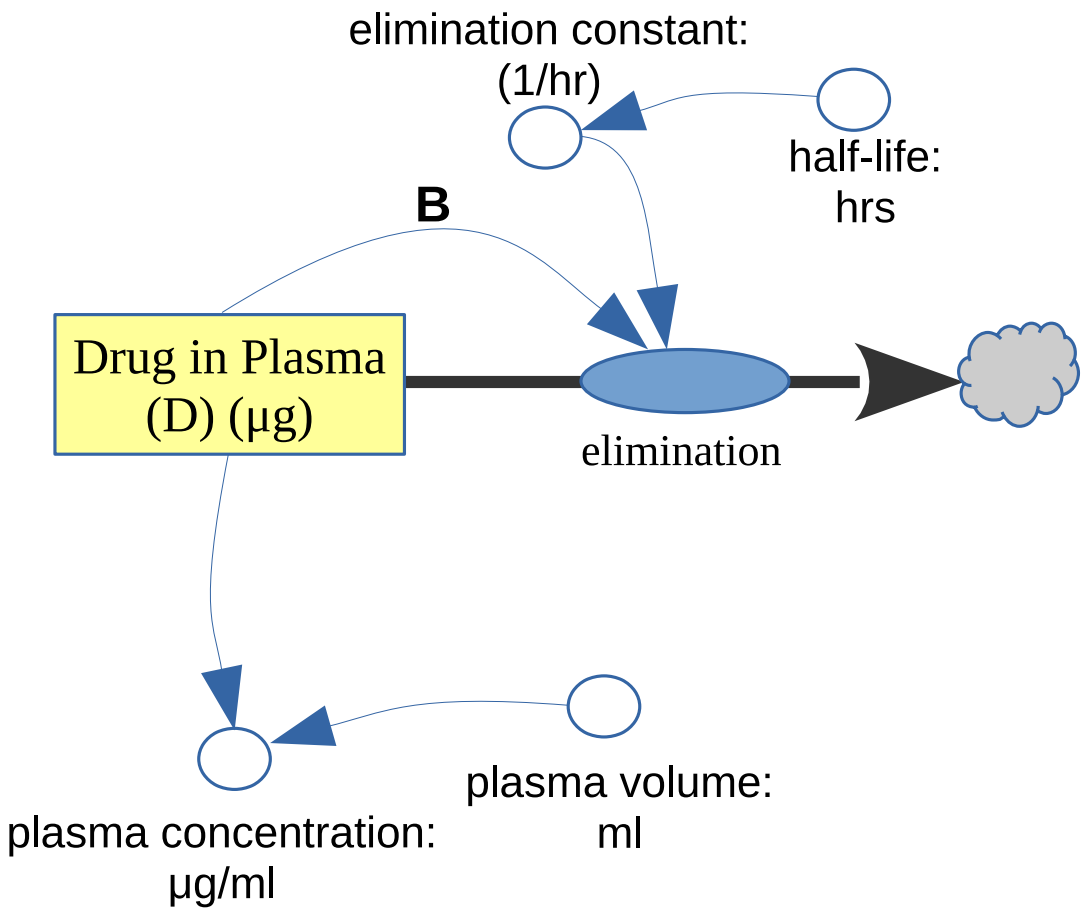
The basic idea behind a computational simulation is to run through a loop a certain number of times, each time simulating one "tick" on a virtual clock. In other words, each time through the loop represents a (short) period of time, during which the important characteristics of the system (stored in program variables) may change. Often we wish to keep track of a particular variable's values over time, so that when the simulation is over we can look back and see its general trend as the simulation progressed. This is an ideal use for arrays since they can store a long sequence of successive values.

To write a Python program to simulate a system depicted in a stock-flow diagram, follow this general procedure:

1. Create a `delta_t` variable to represent a time increment, and an `t` array to represent the time (in units of interest for this problem) for every point in your simulation. (Label the units!)
2. Identify the circles with no incoming arrows at all. If a circle represents a constant (unchanging) value, then create a program variable to represent it. (Label the units!)
3. Identify all the circles whose only incoming arrows are from the circles already defined as variables in your program. Create a variable to represent each one, using the relevant formula. (Label the units!) Keep doing this until the only circles left have incoming arrows from boxes. (These will be your derived stock variables, and will be dealt with momentarily.)
4. Create an array for each stock variable (box). Set the first value of this array to its initial condition (*i.e.*, its value when the simulation begins.)
5. Run through the simulation loop for the specified total time and time increment. Each time through the loop:
   a. Determine the (temporary) values of any flows and remaining ordinary variable(s). Depending on the nature of the circle/oval, this could be (a) an equation based on current values of other circle(s) and/or stock variables, (b) a special value based on the current time (for instance, a "pulse" variable that takes on certain values at specific "clock ticks") or other things.
   b. Create a "prime" variable for each stock, equal to its incoming/outgoing.
   c. Set the next value for each stock (array) to its new value on the next clock tick.
6. For any derived stock variable that has not been defined yet, use a formula to create it based on its incoming arrows. Note that since at least one of these incoming arrows is from a stock variable (which is an array), this circle's variable will also be an array.
7. Finally, plot any array(s) of interest.

Example: Drug dosage (single-dose)

Let's follow this procedure for a specific example. Look carefully at the stock-and-flow diagram below:



Notice that there are *seven* elements in this diagram. One of them — "Drug in Plasma" — is a stock variable, which we'll call "`D`". Three of them — "half life," "plasma volume," and "elimination constant" — are ordinary variables. The first two of these three are ordinary variables because they have no incoming arrows at all. The last of the three (elimination constant) is also an ordinary variable because it has no incoming arrow from a box, only from another circle. One element — "elimination" — is a flow. One element, "plasma concentration," is a derived stock variable because it has an incoming arrow from another box variable. Finally, we have one cloud, which we'll effectively ignore.

Translating this into a Python simulation program is a combination of following the outlined procedure, and using our heads. We begin:

1. *Create a* `delta_t` *variable to represent a time increment, and a* `t` *array to represent the time (in units of interest for this problem) for every point in your simulation. (Label the units!)*

   We always need to decide two basic things about our simulation: (1) how long (in simulated time) will it run for? (2) how much simulated time will elapse between each virtual "tick of the clock?" These two choices are rather arbitrary at this point, but they affect the amount of memory your program requires as it runs, and ultimately, its speed. For now, we'll simulate 8 hours of the patient's body, and set our "granularity" to five minutes:

```
simulation_hours = 8                          # hrs
delta_t = 5/60                                # hrs
t = np.arange(0,simulation_hours,delta_t)  # hrs
```

2. *Identify the circles with no incoming arrows at all. If a circle represents a constant (unchanging) value, then create a program variable to represent it. (Label the units!)*

   The circles with no incoming arrows are "half life" and "plasma volume." Some realistic values for these – depending on the drug and the patient's weight – might be 3.2 hours, and 3000 ml, respectively. Hence, at the top of our Python program, we will define these:

```
half_life = 3.2                               # hrs
plasma_volume = 3000                          # ml
```

3. *Identify all the circles whose only incoming arrows are from the circles already defined as variables in your program. Create a variable to represent each one, using the relevant formula. (Label the units!) Keep doing this until the only circles left have incoming arrows from boxes. (These will be your derived stock variables, and will be dealt with momentarily.)*

   We have one circle whose only incoming arrow is from a previously defined variable: elimination constant. This value describes the rate at which a drug is removed from an organism's system; the formula for it turns out to be $\frac{\ln 2}{t_{1/2}}$ where $t_{1/2}$ is the half-life. So we write:*

```
elimination_constant = math.log(2)/half_life        # 1/hr
```

*Incidentally, don't worry about singular vs. plural when you're writing your comments about units. Here I typed "`hrs`" (hours) for the first few variables and then "`1/hr`" (per-hour) for the next one. Same diff.

4. *Create an array for each stock variable (box). Set the first value of this array to its initial condition (i.e., its value when the simulation begins.)*

   We have one stock variable, so we create a long-enough array for it:

```
D = np.empty(len(t))                          # ug
```

   This line of code creates an array called `D` which will hold *all* the values over time as the simulation runs. Each of the values in this array will have units of μg (micrograms.)[†]

   Then, we initialize the first value of that array to be its value at the start of the simulation. Say our patient took two 325mg pills of the drug:

```
D[0] = 2 * 325 * 1000                         # ug
```

5. *Run through the simulation loop for the specified total time and time increment.*

   As you saw above, the variable `delta_t` (written mathematically as Δt) is our granularity, set to 5 minutes. The t values in this array mean that *every iteration through our loop represents five minutes of time.* Put another way, we will be recomputing the concentration of drug in the patient's blood every five minutes. We now write the loop itself:

```
for i in range(1,length(t)):
```

   This establishes a loop variable called `i` whose value will change each time through the loop. In this case, its values will be 1 through 96. Our simulation will run for 96 clock ticks of 5 minutes each, for a total of 480 minutes, or 8 hours. **Note that the variable `i` will take on values 1, 2, 3, ..., 96.** It will *not* have values $\frac{5}{60}$hr, $\frac{10}{60}$hr, $\frac{15}{60}$hr, ..., $\frac{480}{60}$hr. The latter is the job of the elements of the `t` array, not the `i` variable. `i` does not represent a time value, but simply an iteration number, and an array index. The sequence 5, 10, 15, ... *does* represent the *time* (in minutes) corresponding to each point of the simulation: the first iteration through the loop, when `i`=1, represents the time 5 minutes; when `i`=2, the time is 10 minutes; and so on.

   The reason we start our loop with `i` equal to 1 is that we have already set up the initial condition `D[0] = 2 * 325 * 1000`. In the loop, then, we need to begin by computing `D[1]` based on `D[0]`, then `D[2]` based on `D[1]`, and so forth. Clearly, we need to begin the process with element number 1.

[†]Don't be confused by the "`np.empty`" bit – that just means that we don't need to initialize any of the array's elements with any particular values to start with, since we're going to be writing each value explicitly in our loop anyway. (If this confuses you, use "`np.zeros`" instead, which does exactly the same thing except it makes all the elements of the array zero.)

Now we move on to the body of the loop.

Each time through the loop:

a. *Determine the (temporary) values of any flows and remaining ordinary variable(s). Depending on the nature of the circle/oval, this could be (a) an equation based on current values of other circle(s) and/or stock variables, (b) a special value based on the current time (for instance, a "pulse" variable that takes on certain values at specific "clock ticks") or other things.*

   We have only one flow: "elimination." (Recall that "plasma concentration" is a derived stock variable.) Hence, inside the body of the loop, we compute its value.

```
elimination = elimination_constant * D[i-1]    # ug/hr
```

   This is a pretty standard equation for a balancing feedback loop: Note that we are computing the elimination rate *at step* `i` *of the simulation*. This requires using the amount of drug present in the plasma *at the previous iteration*, which is why we use `i-1` inside the "`[]`" symbols after the array name. This is the common pattern of computing a new value based on a previous value.

   By the way, the units for a flow will always be the units for the stocks it connects *divided by the time increment*.

b. *Create a "prime" variable for each stock, equal to the incoming/outgoing flows it is experiencing.*

   This is normally a pretty easy step: we just take the sum of all the inflows, and subtract the sum of all the outflows. In the drug example, D' is only a single flow: negative elimination. (Why negative? Because the elimination flow is flowing out of, not into, the stock.)

```
Dprime = -elimination                          # ug/hr
```

c. *Set the next value for each array to its new value on the next clock tick.*

   This is pretty much always a super easy step: we use Euler's method to compute the new value of the stock based on its previous value and its rate of change.

```
D[i] = D[i-1] + Dprime * delta_t          # ug
```

   Our super simple model of the body assumes instantaneous absorption of the drug. Therefore, the only thing that affects the amount of drug in the plasma each clock tick is the amount that is eliminated. Note carefully the array indexes in this line of code. We are setting `D[i]` to a value based on `D[i-1]`. This means we are using the *previous* value of the drug in plasma (at array position `i-1`) to compute the *next* value (at array position `i`.)

Also, we multiply by the time increment because each step in the iteration represents a certain amount (in our example, 5 minutes) of time. Hence, the amount of elimination that will occur during a simulated clock tick is, say, five-minutes' worth, and this must be accounted for.

6. *For any derived stock variable that has not been defined yet, use a formula to create it based on its incoming arrows. Note that since at least one of these incoming arrows is from a stock variable (which is an array), this circle's variable will also be an array.*

   At this point, we have accomplished a great deal — nearly our whole purpose. The `D` array now contains values corresponding to the amount of drug in the patient's plasma that was present at five-minute increments over a period of 8 hours. All we need to do now is compute the concentration of the drug over that time, and plot it. First, we compute the concentration of the drug:

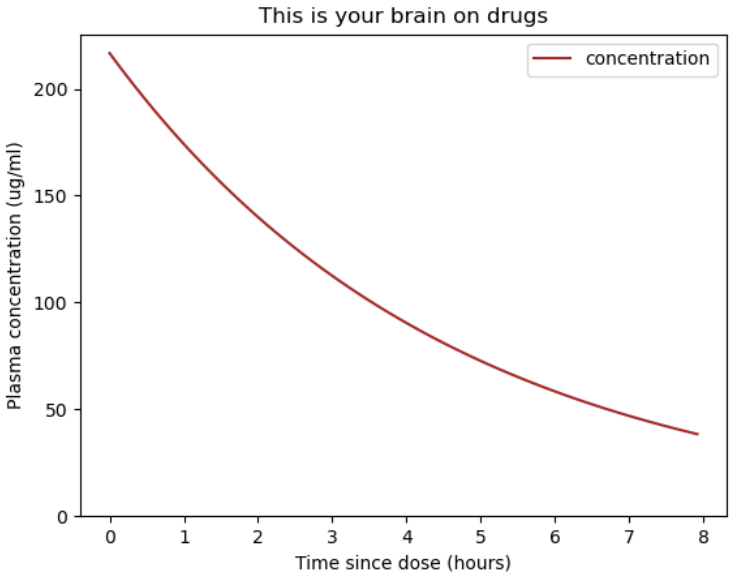
```
plasma_concentration = D / plasma_volume            # ug/ml
```

   using a standard formula. Note that this is a NumPy operation: since `D` is an array (not a scalar), this line of code divides *every* one of its 96 values by the `plasma_volume`, all in one fell swoop.

7. *Finally, plot any array(s) of interest.*

```
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(13,8))
ax.plot(t, plasma_concentration)
```

   Adding labels and ranges is just window dressing.



For reference, here is the entire program:

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

simulation_hours = 8                              # hrs
delta_t = 5/60                                    # hrs
t = np.arange(0,simulation_hours,delta_t)

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

D = np.empty(len(t))
D[0] = 2 * 325 * 1000                             # ug  ("u" = "micro")

for i in range(1,len(t)):
    elimination = elimination_constant * D[i-1]   # ug/hr
    Dprime = -elimination                         # ug/hr
    D[i] = D[i-1] + Dprime * delta_t              # ug

plasma_concentration = D / plasma_volume          # ug/ml

fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(13,8))
ax.plot(t, plasma_concentration, color="brown", label="concentration")
ax.set_xlabel("Time since dose (hours)")
ax.set_ylabel("Plasma concentration (ug/ml)")
ax.set_title("This is your brain on drugs")
ax.set_ylim(bottom=0)
fig.legend()
```