DATA 420 Notes – Feb. 19, 2026

Parameter sweeps

Every simulation has input parameters, from the birth rate of mice to the leakage rate of insulation to the transmissibility of diseases. Changing these parameters changes the simulation's behavior, sometimes in obvious ways, but often not. One of the principal benefits of a simulation is the ability to *experiment* with the parameters and see their effects, both individually and in concert with one another. This is what helps give us a more comprehensive understanding of a complex system's behavior over the myriad of scenarios and environments it may be exposed to.

So far, we've been playing with the knobs and dials in ad hoc fashion. Run the simulation, look at a plot, adjust a parameter and run it again, adjust a parameter and run it again, *etc.* This is exactly the right approach when first exploring a model. You don't know yet what's interesting about it, or where the critical ranges might be.

Eventually, though, we identify a key range of some parameters in which the model exhibits interesting and non-obvious behavior. We'd like to systematically vary a parameter (or two) gradually between certain bounds, so we can get a clear idea of how the simulation behaves across an entire region.

Our solution for this is called a **parameter sweep**. It's just what it sounds like: we're going to "sweep" a parameter (or two) throughout a range of values, running the entire simulation separately *for each value*. Instead of plotting the results of every single simulation (which would be an overwhelming amount of TMI), we need to find a way to quantify something about their outputs that captures the effect we're trying to measure. We can then draw a different kind of plot: not the stock values over time, but the simulation outputs vs. their inputs.

This may seem computationally infeasible, but you'll be surprised how far we can get with just an ordinary laptop these days. In the event where we do face resource constraints, we'll usually manage to get around them with a few simple techniques (principally parallelism).

Step 1: identify an output of interest

To perform a parameter sweep, we'll need to boil an entire colorful, pretty, multidimensional, informative plot down to a single number (or numbers). Yes, this is limiting, but our analysis during a parameter sweep is limited to one aspect of the simulation's behavior. (It doesn't mean we can't separately analyze all sorts of other things.)

Let's take the SIR model as an example. When we ran it, we got a full plot of the S, I, and R stock levels over time. Is there a simpler way to capture some important subset of this information, though, and reduce it down to a number? Sure: *the fraction*

of the population that end up getting sick. This is a number of great interest even in the absence of a plot of the entire dynamics.

In terms of last class's simulation, this is simply:

```
frac_ever_infected = (I[-1] + R[-1]) / (S[0] + I[0] + R[0])
```

(Remember, in Python you can give a *negative* index to an array to count backwards from the end. So while `A[0]` and `A[1]` are the first two values of an `A` array, `A[-1]` and `A[-2]` are the *last* two values. The code above therefore says: "take the last value of the `I` array and the `R` array, and add them together for the numerator.")

So at simulation's end, we add up the currently (`I`) and formerly (`R`) infected people, and divide by the total population. This is easy to quantify, and as we'll see, easy to plot.

There might be other quantities of interest as well, like the number of days it takes the outbreak to reach its peak, or the length of time that 30% or more of the workforce is staying home sick. No matter what the case, we need a way to take the simulation's results and convert them into a meaningful number.

Step 2: parameterize the simulation

The simulation already has "parameters," so you might wonder what this step is needed for. It turns out that mechanically, we're going to want to *put our entire simulation into a (Python) function*, and give this function arguments representing the parameters. I normally name this function "`runsim()`" (for "run simulation.") When called, this function will execute the entire simulation for a set of parameters, and then *return* the output(s) identified in step 1. In our case, we have only one output: the fraction of people ever infected.

Consider SIR. Our simulation looked like this (abbreviated):

```
mean_infectious_duration = 2                # days
transmissibility = .25                      # infections/contact
contact_factor = 5.1                        # (contacts/day)/person

...stuff...

for i in range(1,len(time_values)):
        ...stuff...

plt.plot(time_values,S,color="blue",label="S")

...stuff...
```

We're going to put all this in a function, like so:

```
def runsim(mean_infectious_duration=2,              # days
    transmissibility=.25,                           # infections/contact
    contact_factor=5.1,                             # (contacts/day)/person
    plot=False):

    ...stuff...

    for i in range(1,len(time_values)):
        ...stuff...

    if plot:
        plt.plot(time_values,S,color="blue",label="S")
            ...stuff...

    frac_ever_infected = (I[-1] + R[-1])/(S[0]+I[0]+R[0])
    return frac_ever_infected
```

Notice a few key things:

1. Syntactically, "putting it all in a function" amounts to (1) adding the function declaration ("`def`") and (2) indenting all the code one more tab to the right.

2. The key "parameters" to the simulation (`transmissibility`, `contact_factor`, and `mean_infectious_duration`) literally became *parameters* (arguments) in the Python-function sense.

3. These parameters have default values. (Recall that in a Python function declaration, you can use the "`=`" syntax to provide a default value for parameters that are not given in the function call.)

4. I added a new parameter called "`plot`" which controls whether or not the simulation actually plots something. We will set this to `True` when calling `runsim()` on an ad hoc, exploratory basis, but `False` (the default) when performing a parameter sweep.

5. The return value of the function gives the output(s) of interest. In this case, since the `frac_ever_infected` is the only output, we just returned it as a number. I often find it convenient to return a *dictionary* of values, though, since there are often multiple quantities identified in step 1, and we can thereby easily return several metrics at once.

Step 3: run the parameter sweep

Beginning coders are sometimes surprised when they run the above code and...nothing happens. Realize this is because *simply defining a function does not run it!* You have to actually *call* the function to do that.

So we will. If we want to run the simulation with the default parameters, we do:

```
runsim()
```

If we want to set some of the parameters to different values, we can:

```
runsim(mean_infectious_duration=10, transmissibility=.9)
```

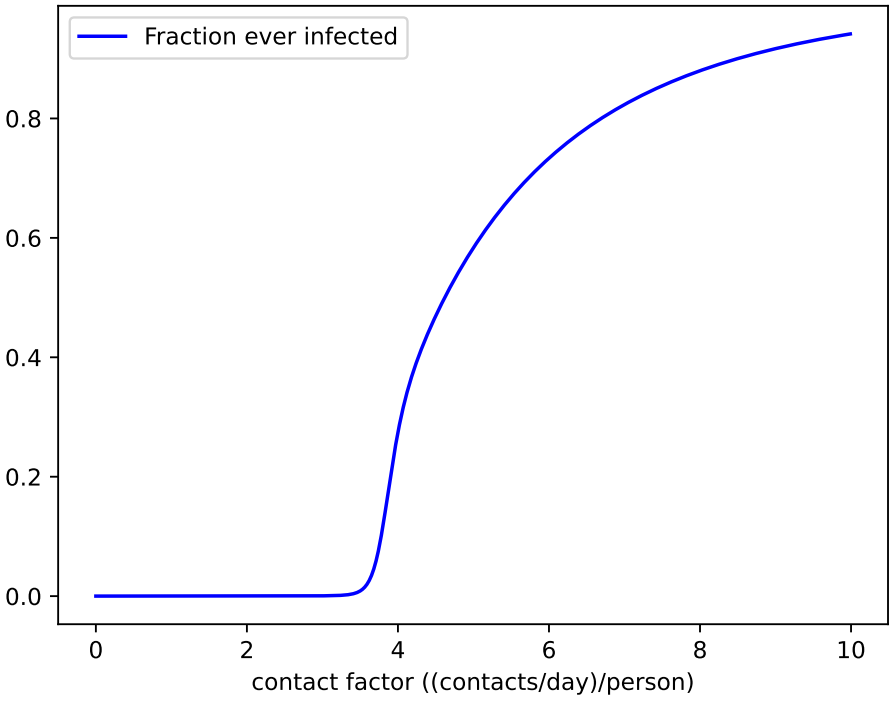
Any parameter we don't specify will use its default value.

So far, there's nothing going on here we couldn't do before. But now's where the magic comes. We're going to define a *range* of values for one of the parameters, and call the function for every value in that range. We'll collect the results as they come back and store them in a vector. And then we'll plot the parameter values versus that vector. Here's the code:

```
...all the previous code, including the runsim() function...

contact_factors = np.arange(0,10,.01)
frac_infecteds = np.empty(len(contact_factors))
for i in range(len(contact_factors)):
    frac_infecteds[i] = runsim(contact_factor=contact_factors[i])

plt.plot(contact_factors,frac_infecteds,color="blue",
    label="Fraction ever infected")
plt.xlabel("contact factor ((contacts/day)/person)")
plt.legend()
plt.show()
```



The "**tipping point**" we mentioned last time simply leaps off the page here. As long as our society is social distancing enough that each person has only 2 or 3 contacts per day, we've prevented an epidemic. But just look at what happens when we get to 4 contacts per day. The explosion is breathtaking – and a stern warning that this model is not *remotely* linear.

Performance considerations

Recognize that we have a loop-within-a-loop here, even though it doesn't look like it. The "`for i in range(len(contact_factors))`" loop is used to run through the sequence of different `contact_factor`s, calling `run_sim()` for each one. The `run_sim()` function, of course, has its own loop inside of it – the same loop we've been writing all semester.

Nested loops like this can be costly, time-wise, depending on the parameter settings. But as I said, you may be surprised how much you can do before it actually matters. I produced the above plot, which triggered 1000 different simulations (since there were 1000 different `contact_factor`s) each of which looped through 1600 different time points. It took only a couple seconds to complete on my laptop.