DATA 420 Notes – Jan. 20, 2026

The units trick

> *"Suppose your car's gas mileage is 35 mpg. You take a 4-hour trip where you average about 50 mph. If gas costs $2/gallon, how much is your fuel cost for this trip?"*

Problems like the above turn out to be easy to solve once you know the trick. The trick is to consider the *units* of the quantities in addition to the number, and always make sure that you combine them in ways that make sense.

The rules are simple:

1. If two values have the *same* unit, you can add or subtract them. Your result will have the same unit. Example: $6\ \frac{\text{hotdogs}}{\text{game}} + 2\ \frac{\text{hotdogs}}{\text{game}} = 8\ \frac{\text{hotdogs}}{\text{game}}$.
2. If two values have *different* units, you **can't** add or subtract them at all. Example: $5\text{ apples} - 3\ \frac{\text{games}}{\text{season}}$ = *(nonsense)*.
3. Curiously, no matter what the units are, you can **always** multiply or divide two quantities. Your result will have *the multiplication or division of the units* as its unit. Example: $3\text{ feet} \times 6\text{ feet} = 18\text{ feet}^2$. Example: $30\text{ dollars} \div 3\text{ pizzas} = 10\ \frac{\text{dollars}}{\text{pizza}}$.

Now in the above example problem, we have four relevant quantities:

$$\frac{35\text{ miles}}{\text{gallon}} \qquad \frac{4\text{ hours}}{1} \qquad \frac{50\text{ miles}}{\text{hour}} \qquad \frac{2\text{ dollars}}{\text{gallon}}$$

We can scratch our head to try to remember and apply formulas. But it turns out that because of the units, *there's only one meaningful way to combine these quantities anyway!* And that's:

$$\frac{2\text{ dollars}}{\text{gallon}} \times \frac{\text{gallon}}{35\text{ miles}} \times \frac{50\text{ miles}}{\text{hour}} \times \frac{4\text{ hours}}{1}$$

Notice how we had to flip (take the reciprocal of) one of the quantities to get the units to "cancel out."

$$\frac{2\text{ dollars}}{\cancel{\text{gallon}}} \times \frac{\cancel{\text{gallon}}}{35\ \cancel{\text{miles}}} \times \frac{50\ \cancel{\text{miles}}}{\cancel{\text{hour}}} \times \frac{4\ \cancel{\text{hours}}}{1} = \frac{2 \cdot 50 \cdot 4}{35}\text{ dollars}$$

A couple of notes:

- Don't worry about singular or plural when canceling out units. "Gallons" is the same thing as "gallon"; that's just an artifact of inconsistent English syntax.
- When a number has no denominator, *it's effectively in the numerator, with a denominator of 1.* (Like in the "4 hours" example.)

It's amazing how much mileage (no pun intended!) we'll get out of this rule this semester. Note that **violating it is always wrong**: you will be guaranteed to get a nonsensical answer as soon as you do.

Rethinking math functions

When I first learned about **functions** in math, the first (and only) thing I thought of was something symbolic, like this:

$$y(x) = \frac{x^3}{1 + \cos x}.$$

Defining a function this way is called expressing it ***symbolically***, *analytically*, or *"in closed form"*.

In fact, I'll let you in on a misconception I had for a long time, which greatly hampered my understanding of math until I had an epiphany one day and defused it. For the longest time, I thought that *all functions had to be symbolic like this.* In other words, I thought that our math teacher was progressively telling us about more and more kinds of functions — polynomials, exponents, trig functions, logarithms, *etc.* — and that the only way to "make" a function was to combine those building blocks the teacher gave us. Looking back, I probably had that misconception because the only examples the teacher ever gave us were built this way. In any event, I thought that unless there were a way to express a function symbolically, it wasn't "really" a function. In fact, I thought the *definition* of a function was a symbolic statement like this.

It's not true. It's not even close to true. In fact, *"most"* functions don't have any symbolic form like this.*† If you carry this misconception, then you risk being hopelessly

*This is imprecisely stated — what does "most" mean if there are infinitely many functions that do have symbolic forms and infinitely many functions that don't? — but I stick by it. The way I think of it is as follows. Suppose you picked any old function at random out of a hat; that is, any old set of pairings from x values to y values. I think you'll agree that it would truly be remarkable if it just happened to work out that the x values and the y values were, at every single point on the number line, related by one of the simple equations we can write down.

†True, there are techniques like the Taylor expansion and the Fourier series that can approximate any

trapped inside the world of symbols and symbolic operations, where life is hard. You have trouble escaping to the world of *numbers*, where life is easy.

Don't get me wrong: equations are awesome. When we have an equation, we know absolutely everything there is to know about the quantities involved. We can instantly compute to unlimited precision how big y will be for a given x, and vice versa. But it's imperative to understand that the above $y(x)$ equation is really just a compact shorthand for saying "when x is 0, y is 0. When x is 1, y is 0.649. When x is π, y is ∞..." Remember: a function is just an assignment of outputs to inputs: a mapping of y values to x values. When this mapping happens to follow a simple and predictable pattern, we can express it as an equation. But if you forget that the equation is just a stand-in for the mapping of y's to x's, you risk losing track of what the equation really means.

A function is an array

I want you to forget all the symbolic stuff for a minute and *think of a function as an array*[‡]. This may seem weird. After all, an array is just a sequence of values: there's no math or other stuff to it. But remember, a function is fundamentally just an assignment of outputs to inputs: you give it an input, and it gives you an output. So we could say that the function y is this:

4, 9, -2, 6, 6, 7, 2.54, 19, 37, 5

All this means is that y assigns the output 4 to the input 0, the output 9 to the input 1, the output -2 to the input 2, *etc.*, and that y doesn't "exist" for any inputs other than 0 through 9.[§]

Now I haven't given a symbolic name to the inputs of y yet. You might expect I would choose the letter x, to be consistent with what we've done so far. But instead, I'm going to choose the letter i. This is because when we think of a function as an array of values, we specify which value we want by giving an *index* to the array. This simply tells us which of the y values to select. When we're talking about these indices, we'll use the boxie notation ("$y[i]$") to designate them. So in the above example, $y[0]$ is 4, $y[1]$ is 9, $y[9]$ is 5, etc. This, of course, is precisely the syntax that many programming languages, including Python, use for arrays.

As a second example, we could say that the function z is defined to be:

---

function by combining infinitely many simple expressions, and that in the limit these infinite sequences approach what might not have been a closed-form expression to begin with. But that's beside the point. I'm talking about concepts here, not approximation techniques, and I maintain that it's a mistake to think primarily of a function *in terms of* symbolic expressions. The symbols' only purpose are as aggregate syntactic shortcuts for relations that are, at their root, numeric and individual.

[‡]In some programming languages, an array is called a "vector." In Python, we often use a *list* for this, although we'll be using NumPy arrays in this tutorial.

[§]As you can see, we're using the Python convention and "start counting at 0."

0, 1, 4, 9, 16, 25, 36, 49

This simply means that $z[0]$ is 0, $z[1]$ is 1, $z[2]$ is 4, *etc.*, and that the only time z makes sense is when i is in the range 0 through 7.

Incidentally, you may recognize that the z function, unlike the y function, follows a familiar pattern: $z[i] = i^2$. I implore you: *do not be distracted by this.* This is just a coincidence, and whatever you do, don't get sucked down into that symbolic hellhole. Just remember that in the vast, vast majority of cases there will be no obvious symbolic representation for our functions/arrays, and in the rare cases where there is one, we'll pretend there isn't.

Now the index i is always a non-negative integer like 0, 1, 2, $\cdots$. The x value represented by that value $y[i]$, however, is *not* necessarily a unitless, non-negative integer. Normally it represents a value in the real world, like "10.5 seconds" or "50 meters." These x values "go" with the i values according to some regular, measured interval. For instance, we might say that a function s measures the speed of a car *at 5 second intervals*. In this case, perhaps $s[0]$ is the speed of the car at some starting time, $s[1]$ is the speed 5 seconds later, $s[2]$ is the speed at 10 seconds after the starting gun, *etc.* We'll call that incremental value "Δx" (pronounced "delta-x") and always keep it implicitly in mind.

Notation-wise, we'll use bananas ("$y(x)$") when we're referring to the value of a function *at a "real-world" x point*. In the car example, $s[0]$ is the same as $s(0)$, since the first data point we have for the function ($i = 0$) corresponds to the starting time for our trip – say, 9:00am sharp. ($x = 9:00:00$). Similarly, $s[1]$ is the same as $s(9:00:05)$, since index number 1 corresponds to 5 seconds after 9am. We could make a table:

i	x	s
0	9:00:00am	0 ft/sec
1	9:00:05am	125 ft/sec
2	9:00:10am	500 ft/sec
3	9:00:15am	1125 ft/sec
4	9:00:20am	2000 ft/sec
$\vdots$	$\vdots$	$\vdots$

I don't want to belabor the obvious, but it's important to keep things straight here. Each value of our function corresponds to both (1) an index, which is simply the number of the element (0, 1, 2, $\cdots$) and (2) an actual x value that corresponds to something in the real world. We could thus refer to our function as either $s[i]$ or $s(x)$, since these are perfectly equivalent: each i goes with one x, so we could use either one to specify an s value. Also notice that the i values don't have any "units" — they're just plain numbers — whereas the x values and the function values do (clock time and ft/sec respectively, in this example.) This will almost always be the case.

Numerical calculus

There are basically two halves to calculus: integration (which gives us an "integral"), and differentiation (which gives us a "derivative.") Let's start with integration.

Numerical integration ("integral")

First, it's imperative to understand what integration *means*. It's so simple it may surprise you. Integration is simply *a running total*.

Let's use the speed example, but we'll make Δx equal to 1 hour. Suppose our function $s(x)$ (or $s[i]$, whichever you prefer) has these values:

$$30,\ 75,\ 75,\ 75,\ 70,\ 70,\ 70,\ 55,\ 30$$

This function represents my car's speed (in mph) on a road trip from Denver, Colorado to Wichita, Kansas, sampled once per hour. For the first hour, I'm driving through town in Denver to get to I-70, so I'm only going about 30 mph. Then, I get to the highway and I can go for a few hours at 75 mph. In Kansas (for whatever reason) they lower the speed limit to 70 mph, and when we get into the Wichita city limits, it's lowered even further.

Now suppose I want to know *how far* I've traveled at each point in time. This is different from speed. In fact, **the distance is the *accumulation of speed***. Think deeply about that for a second. When I drive for one hour at 30 mph, I get "credit" for 30 miles. After a second hour at 75 mph, I've gone a total of 105 mph. A third hour at the same speed yields 180 mph. *Etc.* All I'm doing is calculating a running total.

Numerically, I just want to compute another function $d[i]$ (for "$d$istance") that represents the successive running totals of my mileage. We'll start at:

$$d[0] = 0$$

to represent the fact that initially, at the moment I start my road trip, I haven't gone anywhere. (This is called our "initial condition.") Then:

$$d[1] = 30$$

because $s[0] = 30$, and so I've gone 30 miles after one hour of driving. Continuing on:

$$d[2] = 30 + 75 = 105$$

because $s[0] = 30$ and $s[1] = 75$, so I've driven one hour at 30 mph and another at 75 mph, giving a total of 105 miles. Next,

$$d[3] = 30 + 75 + 75 = 180$$

meaning after three hours I've made it 180 miles. At the end of nine hours,

$$d[9] = 30 + 75 + 75 + 75 + 70 + 70 + 70 + 55 + 30 = 550$$

which is the end of my trip.

Congratulations, you've just integrated. d is now the *integral* of the function s. In a programming language like Python, some sample code would be:

```
s = np.array([30,75,75,75,70,70,70,55,30])
d = np.zeros(len(s)+1)
d[0] = 0
for i in range(1,len(d)):
    d[i] = d[i-1] + s[i-1]
```

(The function `np.array()` in the NumPy library creates an array with a list of specific values; `np.zeros()` creates an array with a certain number of all-zero values.)

Look carefully at the body of that loop. We're just saying: "Make each value of d equal to the previous value of d plus the corresponding value of s." So d follows the simple pattern described above: if s goes 30, 75, 75, 75, 70, $\cdots$, then d goes 0, 30, 105, 180, 255, 325, $\cdots$. Every d value is the *accumulation* of all the previous s values.

Pesky details

A few pesky things you may have noticed about the code above:

1. There are nine speed values, but *ten* distance values. This will always be the case when we integrate. The integrated array needs a starting value (or "initial condition"; in our case, 0 miles), after which it will add/accumulate each of the original values. This gives it a length of one more than we originally had. Do not be alarmed by this.

2. We initialized the `d` array to be a list of all zeros. You might think, "why not just make it an empty list, and append the new value to it each time through the loop?" The answer is two-fold. Conceptually, I like the body of the above loop better because it makes it clear that "`d[i]`" is computed in terms of "`d[i-1]`". This is the essence of what a **differential equation** is, which is mostly what we're going to be using this stuff for. Also, operationally, it turns out to be much more efficient to pre-allocate your entire array of results in memory and then fill in each value as you go, rather than continually appending which will trigger nasty, time-consuming resize operations that will grind our code to a halt.

3. Even after initializing the whole `d` array, we still explicitly set `d[0] = 0`. This was just to make clear what the initial condition was, and to draw attention to the fact that the value of `d[0]` is *not* set inside the loop.

4. Note that loop range, because it's unusual. We start at 1 (not 0) because `d[0]` is special – unlike the others, it's not computed based on any previous value.

About Δx

Now in our example, Δx was exactly 1 hour. This means we were assuming the car's speed was *constant* over each whole hour. So one hour of driving at 30 mph was exactly 30 miles. Suppose we wanted finer granularity for a more precise answer. What if we sampled the speed every *half* hour, instead of every hour? Let's make a new $s[i]$ array as follows:

$$25, 30, 70, 75, 75, 75, 75, 75, 75, 70, 70, 70, 70, 70, 60, 55, 30, 30$$

Everything's the same, except that we have 18 values instead of 9, and Δx is now $\frac{1}{2}$ hour. This means that each interval represents only 30 minutes worth of driving. Therefore, I have to *multiply by* $\frac{1}{2}$ every time I add to the cumulative sum. In other words, after half an hour:

$$d[1] = \tfrac{25}{2} = 12.5$$

I've driven 12.5 miles, and after another half hour:

$$d[2] = \tfrac{25}{2} + \tfrac{30}{2} = 27.5$$

I'm up to 27.5 miles, and after a total of 90 minutes:

$$d[3] = \tfrac{25}{2} + \tfrac{30}{2} + \tfrac{70}{2} = 62.5$$

I'm up to 62.5 miles, and at the end of the whole trip:

$$d[18] = \tfrac{25}{2} + \tfrac{30}{2} + \tfrac{70}{2} + \tfrac{75}{2} + \tfrac{75}{2} + \tfrac{75}{2} + \tfrac{75}{2} + \tfrac{75}{2} + \tfrac{75}{2} + \tfrac{70}{2} + \tfrac{70}{2} + \tfrac{70}{2} + \tfrac{70}{2} + \tfrac{70}{2} + \tfrac{60}{2} + \tfrac{55}{2} + \tfrac{30}{2} + \tfrac{30}{2} = 550.$$

In my program I implement this change by adding a `delta_x` variable of the appropriate size, and multiplying all the d array contributions by it:

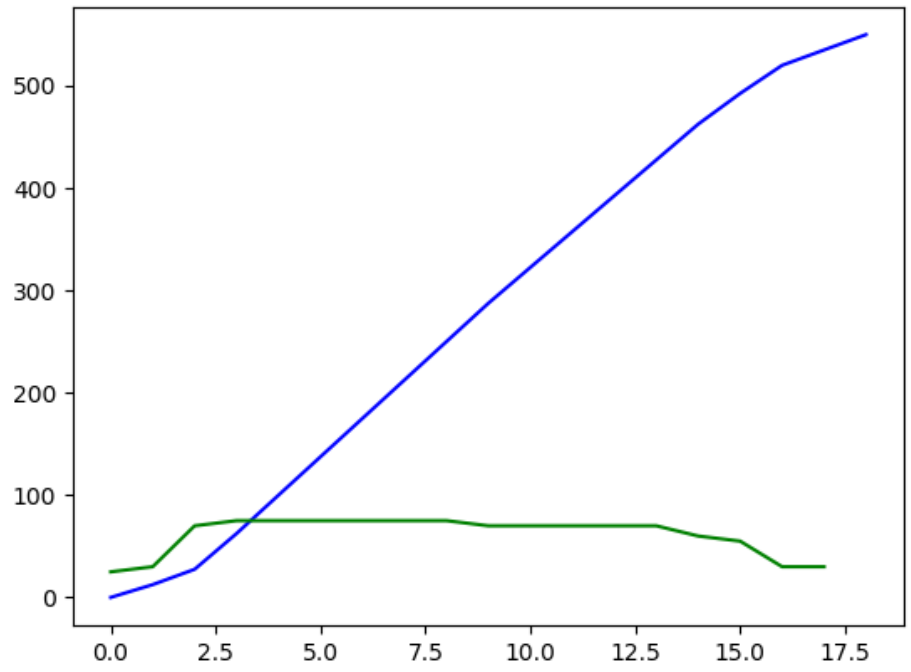
```
delta_x = .5
s = np.array([25,30,70,75,75,75,75,75,75,70,70,70,70,70,60,55,30,30])
d = np.zeros(len(s)+1)
d[0] = 0
for i in range(1,len(d)):
    d[i] = d[i-1] + s[i-1] * delta_x
```

We could plot our results like this:

```
plt.plot(d,color="blue")
plt.plot(s,color="green")
plt.show()
```

Speed is in green (units are in mph) and distance is in blue (units are total miles.) The x-axis is in units of "half hours" since that's what our Δx is. We could of course pretty up this graph by adding more sensible ranges and descriptions. But the important thing to do now is simply to observe that the blue line is basically the running total of the

green line at each value of x. This is what "integration" *means*. After the speed (green line) ramps up and stays constant for a while, the car makes progress towards Wichita and the distance (blue line) accumulates at a constant pace. When the green line tapers off towards the end, the blue line keeps increasing (we don't go *backwards!*), but not as fast.



One other interesting observation is that *integration is a smoothing process*. This means that jaggy, jarring changes in a function tend to be "smoothed out" a bit in the integrated version. Look at the green line in the figure: you can easily tell where the Kansas state line is, and when the speed limit correspondingly drops. But look at the blue line, and you can barely tell that there's a change in angle.

The mathematical way to express all this, by the way, is:

$$d_i = \sum_{j=0}^{i} s_j \Delta x$$

This just means "the distance at a certain index i is *the sum of all the speeds up to that point*, after (of course) multiplying each one by Δx (to take care of the "half hour" phenomenon)." That Σ sign just means "sum." The j is introduced as a sort of "helper variable" for counting purposes since we need i to stand for the upper limit of the sum (and the index of the resulting *d*istance array.)

You may recognize that this is nearly identical to the way we write an integral:

$$d(x) = \int_0^x s(\chi)\, d\chi$$

the main difference being that we are now integrating over the whole continuous range of the real-world x values, instead of at discrete i points. For this reason, too, we use "d" instead of "Δ" to mean "really, really small increments." In our work, we want Δx to be small, because that means finer precision, but not too small, because that means too much memory (and a slow program.) So we'll never *actually* do integrals in the pure mathematical sense. Nevertheless, this is exactly what an integral is: successive addition to get a cumulative sum.

Numerical differentiation ("derivative")

Since integrals and derivatives are in some sense "opposites," and since integrals are basically glorified adding, you're probably guessing that derivatives have something to do with subtracting. Indeed that is the case. If integration is really cumulative summation, differentiation can be seen as *pairwise subtraction*.

(If you've never studied calculus before, take note of the following strange terminology: although we "take the integral" by "integrating," we "take the derivative" by "*differentiating*.")

Derivatives tell us how fast a function is changing at every one of its points. We know that evaluating the function at a particular point tells us the *value* of the function at that point. Evaluating the derivative at that same point tells us *how volatile* that number is: if we scootched over slightly to the left or right of that point, would the function's value be radically different, or only slightly different?

Let's go back to the car trip example. Previously, we started by knowing the speed at each point in time, and we integrated to compute the distance at each point. Differentiation is simply the opposite process: if we start with the position, we can compute the speed.

Suppose we know that the number of miles we have traveled so far at the end of each hour is:

0, 30, 105, 180, 255, 325, 395, 465, 520, 550

In other words, at the end of our first full hour of driving, we've gone 30 miles; after two hours, 105 miles; after three hours, 180 miles, and so on. These are *distances*, so we'll represent them as the function $d[i]$ for $0 \leq x \leq 9$.

To compute the speed at each hour, all we have to do is take one point and subtract the previous point. Consider, for instance, the third hour. At its completion, we have

traveled 180 total miles, whereas when it started, we had only gone 105. Obviously, then, we must have traveled 75 miles during that hour. This gives us a speed of 75 mph, and so $s[2] = 75$. What about the last hour? We started it with 520 miles under our belt, and ended it with 550. So the average speed during that last bit ($s[8]$) was 30 mph. If we crank through the data and compute this difference for each of the pairs of values, we have differentiated the function.

The Python code is again straightforward:

```
d = np.array([0,30,105,180,255,325,395,465,520,550])
s = np.zeros(len(d)-1)
for i in range(0,len(s)):
    s[i] = d[i+1] - d[i]
```

and yields the same array s of speed values we started with in the previous section: 30, 75, 75, 75, 70, 70, 70, 55, 30. Differentiation is the opposite of integration.[¶]

Note that again, we have one fewer value in our `s` array than in our `d` array. This is because if we have n data points in an array, there are only $n - 1$ "differences" we can take, and so only $n - 1$ points in the derivative array. If there are four data points, for example, we can take #3 minus #2, and #2 minus #1, and #1 minus #0, but that's it: only three differences.

As before, one simplifying assumption we've made in the above algorithm is that the units of time are exactly 1 (one hour.) This means that going from 30 miles (at the end of one time step) to 105 miles (at the end of two) means we traveled 75 miles *in one hour*, which then corresponds to exactly 75 mph. Very often we have some other unit than 1, so let's relax this assumption just as we did before. Suppose our total distance every *half* hour was thus:

0, 12.5, 27.5, 62.5, 100, 137.5, 175, 212.5, 250, 287.5, 322.5, 392.5, 427.5, 462.5, 492.5, 520, 535, 550

[¶]Incidentally, the so-called "Fundamental Theorem Of Calculus" blew my mind when I first heard it. If you've studied it, you'll remember that the fundamental theorem states that integration is the inverse of differentiation. But how can "taking the slope of a line" somehow be the exact reverse of "finding the area under a curve?" Those two geometrical notions seemed so different to me that the purported relationship between the two appeared nonsensical. Seen as simply adding and subtracting, however, the relationship is clear. The fact that we got the exact same s array back (above) when we integrated-then-differentiated isn't black magic: it's a simple consequence of the fact that we added, then subtracted. We added up the numbers to integrate, then subtracted each one back off from the cumulative total to differentiate.

All we need to do is *divide by* $\frac{1}{2}$ after doing the subtraction. The reason is that the number of miles between each pair of time steps now corresponds to a *half*-hour of driving, not a whole one. So we have to multiply this pseudospeed by 2 (or divide by $\frac{1}{2}$, same diff) to get a real speed in mph. In Python:

```
delta_x = .5
d = np.array([0,12.5,27.5,62.5,100,137.5,175,212.5,250,287.5,322.5,
    357.5,392.5,427.5,462.5,492.5,520,535,550])
s = np.zeros(len(d)-1)
for i in range(0,len(s)):
    s[i] = (d[i+1] - d[i]) / delta_x
```

Note that we multiplied by Δx when we integrated, but divided by it when we differentiated. This "dividing by Δx" is apparent in the mathematical version of the above code:

$$s_i = \frac{d_{i+1} - d_i}{\Delta x}$$

or just

$$s_i = \frac{\Delta d}{\Delta x}$$

and in the corresponding continuous version:

$$s(x) = \frac{dd}{dx}$$

(Yeah, I know "*dd*" looks weird, but we're using d for *d*istance so just go with it.)

Another way of thinking about it: just as integration is a process of accumulation, differentiation is a process of *anti-accumulation*. By subtracting off $d[i]$ from $d[i+1]$, we are stripping off any accumulation d has experienced up to that point, and preserving only the "immediately new" accumulation — the change that the function has recently experienced.