Differentiation is Easy! Courtesy of A. Ali

I like providing extra credit that is wonky, so... yeah... You're welcome!

In this problem, we'll explore numerical differentiation of noisy data.

For all parts of this question, you may ignore effects of friction and air resistance, assume $g = 10$ m/s^2. You're welcome again! Aren't I generous?

1 Data Generation

- Write a function that takes inputs x, N and outputs

$$y = 2\sum_{k=1}^{N} \frac{(-1)^{k+1}}{k} \sin(kx)$$

  You may use a for loop to implement this, but try to do this as efficiently as possible. The most efficient way to do this won't require you to write a for loop at all, but I don't expect you to be able to do this. You'd have to use `fft()`[1]. Given the output, you should expect x, y to be vectors, and N to be a scalar.

- Your function calculates the Nth partial sum of the Fourier series of a very specific function on the interval $(-\pi, \pi)$. (Which function? It's a secret for now!) Use your function to find the 100th partial sum, evaluated at 1000 uniformly distributed points on the interval. Plot the function (this should be a big reveal!). I prefer using `scatter()` when I'm working with toy problems, but if you prefer `plot()`, be my guest[2].

- This part is going to be rather tedious, but bear with me. Discard the first 100 and last 100 points (Gibbs' phenomena is a pain, you're welcome). Also discard 400 **random** data points of the remaining 800. If you aren't careful doing this, you're going to have to redo this in part 2.

- We're almost done! Add white noise to your data. Do NOT use `awgn()`! Professor Fontaine HATES functions like `awgn(), mag2db()`, etc. SNR of 20 dB is pretty typical, but I feel kind of bad, so 40 dB should do. You're welcome!

That's it! We've prepared some sample data to work with! You all are engineers, so you'll very typically be dealt data that is this disgusting. It's also good practice being able to make toy problems, so I hope you've had fun so far.

2 Getting the Data Ready for Differentiation

- We aren't ready to differentiate yet! To show you this is indeed the case, numerically differentiate the data you have right now, and plot it. Again, I prefer `scatter()`, but whatever floats your boat. What do you see?

  There are a few problems with differentiating the data as it is now. First of all, the data is noisy, and unlike integration, differentiation is NOT a smoothing operation. So what you're looking at right now is noise. Second, we aren't in the position to do anything about this noise either (filtering). Remember when I said discard 400 **random** data points? We should not expect our grid to be uniformly spaced, which is really annoying to deal with. Third–and this may not apply to you–if you weren't careful, you might've lost track of which indices you threw out. If you lost track, you have to redo that one question in part 1 (I really should've numbered them... Oh well...).

  At this point, you should be asking yourself if there's another algorithm that can be used to perform the differentiation. Yes, there are plenty! We've been using the finite difference approach, but there are other ways! But, there are problems with every single scheme that we can potentially use. Here's why: numeric differentiation is an ill-posed problem[3], so it

[1] Gauss actually was the first to develop an FFT-like algorithm. He did this to interpolate the orbit of celestial bodies! Isn't that fascinating? Gauss was a bit of a perfectionist, so he had the habit of not publishing his work (including the FFT-like algorithm), and if he did publish work, he often wouldn't show the steps he took to get his results. This "perfectionism" led him to force his son **not** to enter mathematics or the sciences, for fear he'd "lower the family name." The son immigrated to the United States and I'm pretty sure he became a lawyer. Isn't that depressing?

[2] While the very special function is rather simple, it'll serve as a good toy problem. I made you find the Nth partial sum of the Fourier series because: (1) I thought it'd be funny. (2) It's cool being able to represent functions as infinite sums of sines! (3) In practice, there always is going to be something messed up with your data, so I wanted some rippling here (hence sines are used). We'll make the rippling way worse soon, don't worry.

[3] A well-posed problem is one that (1) has a solution (2) has a unique solution (3) has a solution whose behavior changes continuously with the initial conditions. An ill-posed problem is a problem that is not well-posed.

involves an ad hoc process for choosing a method over another, what parameters to use, etc. So we won't bother with the other algorithms. Instead, as Jacob once told me, we only have a hammer, so everything's got to be a nail. We're going to somehow have to bang out solutions using the finite difference approach. So let's make our data prettier!

- First order of business: we want the data to be uniformly spaced. Interpolate the data onto a uniformly spaced grid from 0 to 1000 (I leave how many samples to your discretion, but you must justify why you made the choice you did) using either `spline()`, `pchip()`, or `makima()` (again, choice is yours, but justify). These are three pretty common interpolators, all use cubic polynomials[4]. Plot your results!
- Look at the spectrum of your signal. Use `fft(x,n)`. We want n to be the nearest power of 2 that is greater than the length of x (for computational reasons, take DSP or ask me later). Plot the magnitude response, and again, do NOT use `mag2db()`. Professor Fontaine will be very sad if you do!

That's it! Now we're ready to try differentiating again! Or are we...? Hint: We aren't.

3 Differentiating

- Not knowing if we're ready to differentiate or not, we differentiate again. Plot this. What do you see?

There are a number of things we can do at this stage. So I'll make you do all of them. Isn't this exciting?

1. Downsample the data, then use a finite difference approach. I'll leave the decimation factor up to you. Comment on potential issues with this method, in particular, about the information content!
2. Fit a polynomial to this data, considering you don't want a low-order polynomial because you lose too much information (I'm not letting you use first-order, is what I'm trying to say), and can't have a really high-order polynomial because you overfit. Differentiate the polynomial, and comment on potential issues with this method. Justify your choice of order!

- Frankly, both methods here suck! For proof, find the second derivative of each data set directly using the finite difference method.

  We can't expect to downsample or fit a polynomial again, we lose way too much information way too fast! What do we do?! We *HAVE* to filter. I won't force you to implement this, because I've had my fun and feel kind of bad... HOWEVER.

- If you had to filter the data, what kind of filter would you use (between E, C1, C2, B), and why? Would you use a high-pass, low-pass, band-pass, or band-stop filter? Why?

If you aren't content, here's some extra stuff you can do (I really want to make this mandatory, but it's a 0-credit course!):

- Read documentation for `ellipord()`, `ellip()`, `cheb1ord()`, `cheb1()`, `cheb2ord()`, `cheb2()`, `buttord()`[5], `butter()`. You don't have to read all of them unless you want to. It'd make more sense to read the documentation for the filter you chose in the previous problem. What parameters are needed, and how would you determine what values to use?
- Look into wavelet denoising! Why would we even consider using it?

[4] Cubic polynomial interpolators are most common, because you've got continuity of first and second derivatives, and don't really run the risk of overfitting.

[5] hahaha.