Lab 5

Learning Objectives:

I. **What characters do I type next?** *How do I interact with data in Python and then display this data?* This lab will focus on using the powerful NumPy, SciPy and matplotlib libraries to be able to mathematically manipulate data and display it using plots.

II. **How do I plan a project and execute my plan?** *How do I integrate Git into my workflow for version control?* This lab will provide the opportunity to practice using Git within a Unix environment to track changes and communicate about them with an external server.

D. **Let me Google that for you.** *How do I use function 'X'?* This lab will require use of many different functions from libraries. One of the most useful skills a programmer can have is the ability to read documentation and understand how to use certain codes. Thus for this lab we want you to look up documentation on functions and figure out how to use them to help you get the results you want.

While you work is about learning to read documentation and apply it to your code.

Part 1 is a warmup for you to play around with some basic but extremely widely used NumPy and matplotlib functions.

Part 2 helps you practice with more complex functionality in NumPy, SciPy, and matplotlib, especially as used for data analysis.

Part 3, if you have time, gives you a chance to learn a tricky function that is a staple of data analysis in Python: `curve_fit`.

While You Work: Google Documentation

It's hard to remember every little detail of a function, so it's useful to be able to look them up. When you need to learn or relearn how to use it up, try Google first. If you figure out that certain search terms work better than others, take note of that for future reference.

Part 1: Playing with simple arrays and plots

This part is for you to start playing with NumPy and matplotlib. Open file `trig.py`. Notice that we already included the relevant `import` statements so that you can immediately begin to use functions from the NumPy and matplotlib libraries. (Note that we imported them as `np` and `plt` so you can access their functions by typing `np.function()` and `plt.function()` rather than the full library name.)

What's the name of the NumPy function for *sinx*? Make an array of x-values from 0 to π with a spacing of 0.01 using the `np.arange` function. Use it and the `plt.plot(x, y)` function to plot *sinx* on the interval $[0, \pi]$. Does it look like what you expect?

Now write a function `integrate(y, dx)` that takes an array of y-values and a step size dx of the corresponding x values and numerically integrates the function. You do not need the x-values of the function for the integration. *Hint: an integral is an infinite sum; if we have discretized values instead of a continuous function we need to revert to a finite sum. This function can be written in one line.*

Integrate your `sin(x)` function over the same range and compare to analytically known results (the integral of $sinx$ from 0 to π should give 2). How close can you get? How does it depend on the spacing dx?

Repeat the above plotting and integration for $cosx$. Does everything still work? (You should be generating two plots: one for $sinx$ and another for $cosx$.)

Instead of using your `integrate(x, y)` function, find and use a NumPy function that also numerically integrates your function. Which function is it and how do the results compare?

Part 2: Data Analysis

Open up the script `data_analysis.py`. You'll notice that it contains a function `noisy_packet()`. This function creates a Gaussian wave packet with some simulated experimental error. This function accepts four arguments: an array of x-values, a wavenumber k (for the sinusoid $sin(kx)$), a standard deviation σ (sigma, for the Gaussian), and a parameter `noise_amplitude` (how much simulated error you want). It produces an array of y-values describing a Gaussian wave packet with those parameters. **Your task is to clean a lot of the noise out of this function using a Fourier transform.**

- **Wave packet:** a wave packet is a function that is sinusoidal within a small range and pretty much zero everywhere else. It is created by multiplying a sinusoid by an "envelope": some decaying function—in this case a Gaussian, $e^{-x^2/2\sigma^2}$, where σ is the standard deviation—which selects the portion of the sinusoid to keep.
- **Gaussian noise:** To simulate experimental error, this returned noisy packet has Gaussian noise. This means each "measured" y-value deviates from the true (wave packet) y-value by some amount, and that the deviations are Gaussian distributed: it is very likely that most deviations will be close to 0, but there's a small probability that any particular deviation could be very large (positive or negative). The Gaussian noise from this function has standard deviation `noise_amplitude`.
- **Fourier transform:** Conceptually, the Fourier transform accepts a function of x (e.g. a wave packet) and returns a function of k, which tells you how strong the mode with wavenumber k is in the provided function. Its inverse takes the function of k and returns the original function.

Now write code in the `main` function. From `main`, call the `noisy_packet` function with different parameters and plot the resulting noisy packets. Produce a noisy wave packet with a `noise_amplitude` of 0.2, a wavenumber of 5 and a standard deviation of 1. What do you expect it to look like? Plot it; does it look like what you expect?

We'll now do some simple data cleansing to get rid of the noise. Fill in the function `clean_data` with the following steps. First take the Fourier transform of your noisy data to get the transformed data and the frequencies (*Hint: look at* `numpy.fft.rfft` *and* `numpy.fft.rfftfreq`). Now, to clean the data, figure out a way to zero out the transformed data corresponding to noisy high frequency components. Inverse Fourier transform the result using `numpy.fft.irfft` to generate your "clean" data. Plot the cleaned data along with the noisy data. Have you successfully cleaned up the wavepacket?

Part 3: If You Have Time

Now fit a gaussian wave packet with parameters k and σ to the cleaned result using `scipy.optimize.curve_fit`. How close are the best fit parameters to your original k and σ? *(Hint: you will need to define a function that returns a Gaussian wave packet to pass as a variable to* `curve_fit`*).* Also try using `optimize.curve_fit()` on your original `noisy_data`. How do the k and σ values obtained from the noisy data and the clean data compare? Was there an easier and more efficient way of cleaning up your data than using the low-pass filter in part 2?