Lab #6

Physics 91SI Spring 2013

Objective: In this lab, you will familiarize yourself with numpy, scipy and matplotlib, the 3 essential components of the python scientific library. You'll get to use them for both data analysis and modeling.

As usual, log on to `corn` and clone over the starter repository:

```
hg clone /afs/ir.stanford.edu/class/physics91si/src/lab6 lab6
```

Remember to `hg commit` often to save your changes, and submit your code at the end of the lab. Also, do not be afraid to use the python debugger `pdb` in this lab! Recall that you can run `pdb` on a python script as

```
python -m pdb <script name>.py [arguments]
```

Part 1: Playing with simple arrays and plots

This part is just a brief introduction; don't spend more than about 10-15 minutes on it. Ask for help if you need it. To begin, open an ipython shell in pylab mode using

```
ipython --pylab
```

This will allow you to use all scipy, numpy and matplotlib functions without referring to the packages (i.e. `numpy.array()` becomes just `array()`).

Make an array of x-values from 0 to π with a spacing of 0.01 using the `arange()` function. Use it and the `plot(x,y)` function to plot a couple of functions (don't forget to call `show()` after you call `plot()`!), such as `sin(x)`, `exp(-x)` and maybe the first spherical Bessel function, `sin(x)/x`, on the interval $[0,\pi]$. Do they look like what you expect? Why do you have to restrict your range for the Bessel function? Play around a little bit.

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.*

Compare your solution to analytically known results (e.g. the integral of $\sin x$ from 0 to π should give 2). How close can you get? How does it depend on the spacing dx?

Part 2: Simple Data Analysis

Open up the script `data_analysis.py.` You'll notice that it contains an incomplete function `noisy_packet()`. This function creates a **gaussian wave packet** with some simulated experimental error. (A wave packet is a sinusoid that has an amplitude that is maximum at a certain location and decays in all directions from there. It is created by multiplying some decaying function—in this case a gaussian, $e^{-x^2/2\sigma^2}$, where σ is the standard deviation—by the desired sinusoid function.)

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.

To simulate experimental error, add in **gaussian noise**. This is a function which creates a gaussian, picks a random point in the area underneath it, and returns the x-value (so a particular value it returns could be anywhere between infinity and minus infinity, but it's more likely to be close to 0). The gaussian noise should have standard deviation `noise_amplitude`. *Hint: consider the function `numpy.random.randn()`.*

Now write a `main()` function. From `main()`, call your `noisy_packet()` function to produce a noisy wave packet with a `noise_amplitude` of 0.2, a wavenumber of 5 and a standard deviation of 1. Plot the function. 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 "data" (the noisy gaussian) using `numpy.fft.fft()`. Shift the low frequency components to the middle of the array using `numpy.fft.fftshift(array)` (remember to make sure your frequency array and your Fourier transform array indices match up!). Plot the absolute value of the result. Does it look like what you expected? You should be able to see come clear peaks in the low frequency region and a bunch of noise in the high frequency region. Now, to clean the data, figure out a way to zero out the noisy high frequency components (but not the information-carrying low frequency components). Inverse Fourier transform the result using `numpy.fft.ifft()` and plot its real part. What does it look like? Compare it with the original data.

While you run and re-run this program, you might want to comment lines of code that produce plots that you've already seen.

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?

Part 3: A Simple Simulation

You might have heard of the wave equation. (If you haven't, keep reading.) In this part, you will write your own simulation that models the 1D wave equation on a finite interval. Open up the script `wave.py`. This script uses an array of x and y values and models $y(x,t)$ as if it obeyed the wave equation (with wave velocity c)

$$\frac{\partial^2}{\partial x^2} y(x,t) = \frac{1}{c^2}\frac{\partial^2}{\partial t^2} y(x,t)$$

Instead of writing derivatives, which represent infinitesimal changes, we can rewrite this equation in what's called **finite difference form**. Then we can solve it for the i'th y value, y_i. This allows us to step the equation forward in time. Ask for help if you are interested in the details. The update rule for the i'th y value is given to you. We have also already written the function `init()` for you as well as a skeleton for the functions `simulate()` and `update()`. Using the comments in the code as guidelines, complete the program such that `simulate()` loops for a given number of time steps, and at every time step

- updates the new y values
- makes a plot of the current y values

Remember that while running a script, you can terminate the execution using Ctrl + C.

If your simulation works, play around with it a little bit. Implement different initial conditions. Which initial conditions cause the gaussian to move in just one direction instead of splitting up? Use other functions as initial conditions. Can you produce standing waves? Can you implement Neumann boundary conditions? How? What happens if you make the time step or the wave velocity too large?

Challenge: Modify the code such that it simulates the **diffusion equation** with **periodic boundary conditions**. This only requires modifying a few lines in the code!