IC/HW 3: Programming Assignment

Due: 12:01 AM, Thurs 10/11.

For each question (but not each sub-part), create a SEPARATE FILE. Within each file, use one and only one cell (input box), and use comments to indicate different sub-parts. For each question, please print out your script and the resulting output to hand in.

Rule for coding in this class: YOU MAY NOT USE MORE THAN ONE CELL PER FILE. There are a few reasons why: (1) Keeping clean, concise, readable code is essential for avoiding mistakes and debugging; (2) When doing real programming there's no such thing as cells (the cells belong to the interactive interface Jupyter, not to Python); (3) Because of the way Jupyter runs and remembers code, debugging with multiple cells in Jupyter is a nightmare; (4) Using multiple cells can cause code that would normally cause an error, to instead work but give a wrong answer.

If you ask for help and your file has multiple cells, we will tell you to fix it first.

1. **Warm-up.** (10 pts).

(a) Print "hello world".

(b) Import numpy as np.

(c) Create a variable $x = 2$. Then set $x = x^2 + 1$. Print x.

(d) Use np.linspace to create a numpy array called y which takes values from 0 to 10 using 11 points. Print `y[0]`. Print `y[1]`. Print `y[-1]`. This is the basic syntax for addressing values in an array. Now print `y[1:3]`. Print `y[3:]`. This is the basic syntax for taking sub-arrays from an array.

(e) Boolean masks allow you to isolate certain values of an array. Use the same variable y from the previous part. Type `mask = (y>3)` and print *mask*. Print `y[mask]`. To do this without a middle step, print `y[y>3]`.

(f) Functions of arrays are applied to each value. When multiple arrays are present, Numpy tries to put them together in the obvious way. When all arrays in the equation are the same length, functions will be evaluated element-wise. Let `a=np.array([0,1,2,3])` and `b=np.array([0,10,20,30])`. Print `2*a+a*b`. What happens if you change the length of b (no response needed)?

(g) String formatting of numbers is useful for the creation of text files and printed results. Let $x = 21.12345$. Print `"%d"%(x)`. Print `"%.3f"%(x)`. Print `"%.4e"%(x)`. Print `"%05d"%(x)`.

(h) Write a function called `myfunc1` which takes in a number x and returns x^2. Print `myfunc1(3.3)`.

2. **Write a linear regression function.** (30 pts). Read the *Python Plotting Tutorial – Part 8* online. The tutorial gives a method for calculating the slope and intercept of a best fit line, given data with known error bars, along with the uncertainty of the resulting slope and intercept. Your task will be to implement this function.

(a) Write a function called `linfit`. This function should have inputs `(xdata, ydata, yerror)`. Each of these inputs should be a numpy array. It should output an array of four values, in the form `np.array([slope, intercept, sigma_slope, sigma_intercept])`. (*Hint: Use np.sum to evaluate U_n and W_n*).

(b) Apply your function to the input data

```
x = np.array([ 0., 1., 2., 3., 4., 5.]),
y = np.array([ -7.1, -11.9, -16.8, -22.4, -26.7, -31.8]),
yerr = np.array([ 0.3, -0.2, -0.7, -1.2, -1.7, -2.2]).
```

Extract your results, and use a command like

```
print("results")
print("slope = %.3f +/- %.3f"%(slope, sigma_slope))
print("intercept = %.3f +/- %.3f"%(intercept, sigma_intercept))
```

to print the result out nicely.

(c) Create a plot of the data with the best fit line. Make sure to include title and axis labels, a grid, and a legend. The data points should use a '.' marker with vertical error bars. The best fit line should be a smooth line, and should extend over the interval $x = (0, 8)$. The data points should lie on top of (as opposed to behind) the line. Save your figure as a pdf.

(d) Calculate χ^2 and $\bar{\chi}^2$ for your fitline. Print these out nicely, like above.

(e) For this part, use the given `y` data only, you can ignore the `x` and `yerr` data. For the `y` data, calculate the mean $\bar{y}$ and standard deviation s. Print results nicely.

(f) Now include both the `y` and `yerr` data. Calculate the weighted mean (with weights $w_i = 1/\sigma_i^2$), and the uncertainty of the weighted mean (see stats hw). Print results nicely.

3. **Plot the complex impedance.** (20 pts). For a completely parallel RLC circuit (not the one you have in lab), the complex impedance as a function of frequency is given by

$$z = \left(\frac{1}{R} + \frac{1}{i\omega L} + i\omega C \right)^{-1} ,$$

where $\omega = 2\pi f$. The easiest way to plot the magnitude and phase of this function is to evaluate it as a complex function in Python, then numerically take the magnitude and phase. *Python Plotting Tutorial – Part 3* is a useful reference for this exercise.

(a) Write a function which takes in an array of frequency values f, and returns an array of complex numbers $z(f)$, according to the formula above. Your function should also take the parameters R, L, C as inputs. (*Hint: The complex unit in Python can be written as* `1j`).

(b) In separate figures, plot the magnitude and phase of the complex impedance over the frequency range $f = 1Hz$ to $20kHz$. First use the parameters $R = 1k\Omega$, $L = 10mH$, and $C = 100nF$, using a red line. Then with a blue line, overlay the same graphs, but with the resistance doubled. Format, label, and display the graphs appropriately. Save your plot as a pdf.

4. **Basic if and for loops.** (10 pts). For basics of `if` and `for` loop syntax, see the official Python documentation at https://docs.python.org/3/tutorial/controlflow.html.

(a) Write a script which produces a list of every prime number between 0 and 100.

Some useful tips:

– The modulus `n % m` returns the remainder of n/m.

– The boolean equals sign `==` should be used in `if` statements.

– `for` loops in Python iterate over a list.

– The function `range(int)` returns a list of integers from 0 up to (int-1).

– You might start by writing a function which decides if a given number is prime, and returns either `True` or `False`. One way to do this is to use a boolean variable which can be changed from `prime=True` to `prime=False` when certain conditions are met.

BONUS. **Visualizing the helium spectrum.** (+10 pts). Matplotlib provides a diverse array of tools to create data visualizations using surprisingly simple commands. Can you recreate this visualization of the first order helium spectrum? You can use data from the file helium_lines.txt (link), which is a text file written in Python language.