PHYS 2310H, Assignment 4

Due Thursday, March 19.

1. Read Appendix E.4 of Kinder and Nelson on the topic of Scope of a variable. It goes into more detail than I did in class. Then consider the following code (adapted from Kinder and Nelson).

```
def outer():
    x, y = 'E', 'E'
    def inner():
        x = 'L'
        print("Inner Function:", x, y, z)

    inner()
    print("Outer Function:", x, y, z)

x, y, z = 'G', 'G', 'G'
outer()
print("Main Program:", x, y, z)
```

   (a) What is the output of this code? Explain why, in terms of the scope of each variable.

   (b) How does your answer change if `x` is declared to be `global` at the start of the function `outer()`? Explain why, in terms of the scope of `x`.

   (c) How does your answer change if `x` is declared to be `global` at the start of the function `inner`, but not in `outer`? Explain why.

   (d) The function `inner` is an object, just like the variables `x, y, z`. If you try to access it by typing `inner()` in the console, you'll get an error message saying that it's not defined. That's because the function belongs to the `outer` function. How could you add a `global inner` statement to the code to make the function accessible? **Caution:** You need to `%reset` your code every time you try something new because iPython will remember previous `global` declarations.

2. This question has to do with Appendix E.4.2 in Kinder and Nelson. Consider the following code, which is meant to display unintended side effects of functions. A **side effect** is when a function changes the value of an argument. Generally, we want functions not to have side effects (you wouldn't be happy if taking `sin(x)` overwrote the value of `x`).

```
import numpy as np
def f(x):
"""
This function doesn't change the value of x in the main program
"""
x = x + 10
return x

def g(x):
"""
This function changes the value of x in the main program.
```

```
"""
x += 10
return x

x = np.arange(10)

print("f(x) = ", f(x))
print("x = ", x)

print("g(x) = ", g(x))
print("x = ", x)
```

Why is it that `f(x)` does not change `x` in the main program, but `g(x)` does? *Hint:* You will find it useful to use the `id()` command to keep track of what Python is actually doing. There are two key lessons here:

- there is a fundamental difference between using assignment (i.e. `=`) and using a built-in method (which is what `+=` does).
- pay close attention to how you handle variables. The `id()` function can be really useful to see what's actually going on.

3. In this question, you are asked to model a damped, driven, linear harmonic oscillator, and to compare your numerical solution with the exact solution.

   (a) Prove Eq. (3.8) in G& N. Then explain why Eq. (3.8) means that the energy grows linearly with the total time of the simulation.

   (b) Now we come to simulating the pendulum. The first part of this question should contain a page or two describing the assumptions that you make and/or values of variables that don't change throughout the question.

   (c) Write a code that solves the problem of the damped linear harmonic oscillator (no driving force just yet) using the Euler-Cromer method.

   (d) Make a figure that shows your numerical results as symbols (e.g. circles, squares, etc.) and the exact solutions as lines for the three cases shown in Fig. 3.4 of G&N. To make the figure legible, you may want to plot every 10th data point. Plotting the results this way allows the reader to see just how good the fit is or isn't.

   (e) Next, for the underdamped case, make a figure showing the difference between the exact and numerical solutions. How small do you need to make dt to make the largest error less than 10^{-4} radians?

   (f) Now for the main part of the problem. Numerically investigate the damped, driven pendulum as a function of Ω_D, and confirm the existence of a resonance at $\Omega_D = \Omega$. Compare your results to Eq. (3.16) *quantitatively* by plotting the equation on top of the amplitudes you get from your numerical simulations.

   For the last part of the problem, you'll need to provide a discussion of how your removed the transients from your analysis, and how you obtained the amplitudes. **Remember:** If you are doing a calculation multiple times with different parameters, then put the calculation in a function and call that function multiple times.

4. This question is based on Sec. 3.3 of G & N. I want you to analyze the evolution of the damped, driven, nonlinear pendulum for driving forces between $f_d = 1.4$ to $f_d = 1.5$. Other parameters should be taken the same as in the textbook. This question has several parts:

   (a) I want you to write a `module` that contains the function `pendulum_EC()`. The function should take an array of times as an argument (plus anything else that's needed) and return two arrays, `theta` and `omega`. If the module is called `lyapunov.py`, then you can import it using `import lyapunov` or (for example) `import lyapunov as lp`. Then, you can access the function using `lyapunov.pendulum_EC()` or `lp.pendulum_EC()`. Note that variables defined in the module have local scope within the module, although they can be accessed using the appropriate module prefix. For example, if you define `g=9.80` as a variable in the module, you can access it via `lp.g` from the main code.

   (b) The main part of your code should be in a different file (e.g. called `lyapunov_main.py`). Make a few (two or three) representative figures showing how the pendulum trajectory evolves in the different physically important regimes for $1.4 < f_d < 1.5$. Note that you may want to map $\theta(t)$ back into the interval $-\pi \leq \theta < \pi$, which you can do using the `np.remainder()` command. Your caption should explain what's going on.

   (c) Now, make figures like Fig. 3.7 in G & N showing how the Lyapunov exponent changes with f_d. A few things might be helpful for you:

   - The smaller you make the initial $\Delta\theta$, the larger the time window over which you can extract λ.
   - Make sure `dt` is small enough that you can get reliable values for λ.
   - I suggest that you use `np.polyfit` on the logarithm of $|\Delta\theta|$ to get the Lyapunov exponent. You don't want to fit over the entire range of motion, but only over the range over which the motion is logarithmic. Remember that numpy arrays can take Boolean arguments, so `theta[t<5]` is a slice of the array `theta` corresponding to parts of the array `t` that are less than 5s. See Sec. 2.2.10 of Kinder & Nelson for a refresher.
   - Rather than shown dozens of figures, show a few representative figures and then make a graph of λ versus f_d.
   - It might seem easiest to write a loop over f_d values and automatically generate the data; however, you'd better be sure that you can fit over the same range of times for each case if you do that.