Python Waves Lab

PHY152

This lab is to be done in pairs. You will have time in lab to complete it. If you do not finish it in lab, you can hand it in by email before your next lab session (1 week).
Please have your names and tutorial group in a comment at the top of the code.
Make sure to save your code to a USB stick or email it to yourself since you can not later retrieve files you save on the lab computers.

Computational Learning Goals

- The purpose of this lab is to learn about and use basic concepts in python programming, as well as using python to make visualizations

- Concepts include: mathematical operations in python, arrays/lists, loops, functions, external python modules, plotting graphs, and animating graphs.

- Each of the questions in this lab will present some information on which python programming concepts to use, and some example code. You will, however, need to write the programs completely, since no starter code is provided.

- You may want to visit the physics department's compwiki page for python resources if you do work outside of the lab: http://compwiki.physics.utoronto.ca/

- Note that in this lab we use the word 'list' to talk about something that is also called an 'array' in other courses and programming languages. In python, the actual type is called 'list', and so we will use that name.

Physics Background

- **Wave functions:** A wave function is a mathematical description of a wave. We can describe a 1-dimensional wave as:

$$y(x,t) = A\sin(kx - \omega t + \phi)$$

where A is the amplitude, k is the wave number, ω is the frequency, and ϕ is the phase shift.
The equation above is a function of position and time.

- **Beats:** ‘Beats’ is a wave phenomenon that can occur when two interfering waves (moving in the same direction) have the same amplitude and phase, but have slightly different frequencies.

- **Standing Waves:** Standing waves are produced as the result of the interference of two waves of identical frequency and amplitude while moving in opposite directions. All standing wave patterns consist of **nodes** and **antinodes**.

Lab Instructions

For grading purposes please hand in the following parts. Add the last name of one of your group members to the end of the files as indicated.

- **Q1**: Submit your code as a file called `beatsLASTNAME.py`
- **Q2**: Submit your code as a file called `standingLASTNAME.py`

Do all of the parts marked **Lab Task** in each question.

The parts marked *Example* are meant to teach you python, but they **should not** go in your code.

Ensure you comment your code. This will make it easier for the marker to assess your code. You do not need to comment every line, but give a short explanation of what your code

```
# Comments are made with a hash symbol
```

Q1: Beats

Importing modules

Open up a new python file. Do all the work for Q1 in this file.
In order to model a wave function, we will be using python functions from external modules. Python doesn't have built in trigonometric functions like sine and cosine, so we need to add them to our program. We do this by using the **import** keyword in python.

Lab Task: At the top of your python file, add the line:

```
from numpy import *
```

This takes all of the python functions in the module 'numpy' and adds them to your python program. You can now use functions like **sin** and **cos** in your program.

Defining functions

Functions are very useful in python since they allow you to minimize repeated code and give you an efficient way to see what would happen if you change the values of certain variables.
A function is made up of its **input parameters** and its **function body**.
In python, functions are defined using the def keyword. The following example goes through a function definition (*remember that examples do not have to go in your code*).

Example 1: *Define a function called 'addTwo' that takes an input parameter called 'x' and returns 'x+2'. Then call the function with the parameter 1.*
Solution:
In python, functions are defined using the def keyword as so:

```
def addTwo(x):
    return x + 2

answer = addTwo(1)
print answer
```

In the above example, the **input parameter** is 'x', and the **function body** is 'return x+2'. This is a simple function, and in general, function bodies can be longer. Note that anything in the function body **must be indented**.
You should, of course, get the number 3 returned to you (the `print` statement writes the variable `answer` to the python console).

You will now create a function in your own code.

Lab Task: Define a function called **sineWaveZero** that takes in the correct input parameters to return the result of the expression

$$A\sin(-\omega t)$$

Having this function in your code lets you call it to model any sine wave function at position 0 with a 0 phase shift (hence the 'Zero' in the function name), at any time 't'.
You will need more than one input parameter for your function, since more than one part of this expression can vary. You can do this by separating the arguments by commas:

```
def moreThanOne(one, two, three):
```

Remember that you already have the sin function imported.
When you are done, test your function to make sure there are no errors in the code. Use some test values to ensure it works as expected.

Plotting graphs

In this part of the lab, you will use the function you created above to plot graphs of two wave functions (y vs. t), and the result of how they interfere.

In order to make a plot in python, you will have to import another module. You will do it a little differently this time.

Lab Task: At the top of your python file, under the first import statement add the line:

```
import matplotlib.pyplot as plt
```

This imports the plotting library functions into a variable called 'plt' that you can use anywhere in your code.

To make a plot, you need to have lists for your **independent variable** and **dependent variable** (the result of your function).

First, you make the independent variable list.
Since computers can not directly handle continuous functions (we can't have infinite-sized lists), you must choose how many points you want to evaluate your function at. One way to do this is to use the 'arange' function from numpy (an example is shown below).

Once you have the independent variable list, you can put it as an input parameter to a function, and you will get back a dependent variable list with the result of calling the function on each element of the independent variable list. See the example below.

Example 2: *Make an list 't' of numbers from 0 up to and not including 2, with a step difference of 0.5 between each number. Then make a list y of the result of cos(t).*
Solution:

```
dt = 0.5
t = arange(0, 2, dt)
y = cos(t)
```

It's good practise to create a variable 'dt' for the step difference.
You can now create one for the lab.

Lab Task: Create an independent variable list 't', and two dependent variable lists that call your **sineWaveZero** function on t.

Make a 0.1 step difference for t, and make the list from 0 up to and not including 100.
For both dependent variable lists, make sure the amplitude is the same. However, make the frequencies somewhere between 4-6 rad/s, with a difference of about 0.2-0.4 between them.

Now that you have all the data for a plot, and you can now make the visual graph of the data. This example goes through plotting a function represented by the variables **t** and **y**, where t is an independent variable list, and y is a dependent variable list. When this code is run, a plot of the function would open in a new window.

Example 3: *Plot a function y(t) and show the plot.*
Solution:
First, create a ***plot figure*** *variable.*

```
fig = plt.figure()
```

Note: figure() is a 'method' of the variable plt. Methods are discussed later in this lab.

Now make a plot (or 'subplot' as it is also called) in the figure that takes up the entire figure window. The '111' just makes the plot take up all the space in the figure. This would change if we want to have 2 different plots in the same figure, for instance.

```
subplot = fig.add_subplot(111)
```

Add the function to the plot. You can add more than one function to the plot.

```
subplot.plot(t,y)
```

Show the plot figure.

```
plt.show()
```

You will now make plots for the variable lists you made earlier.

Lab Task: Create a plot figure and make three plots.
The first two plots are of the two sinWave functions.
The third one is of the sum of the two functions.
Hint: You can get the sum of two lists **a** and **b** element-by-element by simply writing `a + b`

Show the plot figure. This represents a 'history graph' of the two waves and their sum at position x = 0.

After running your code, you should see three plots in the same figure. One, which is the sum of the two functions, should exhibit a beat pattern.

You many need to tweak the frequencies in your functions if you don't see a good beats pattern.

Make sure to save your file as `beatsLASTNAME.py`

Q2: Standing Waves Animation

Open up a new python file.
Copy the import statements from Q1 function into your new file.

Lab Task: Define a function called **sineWaveZeroPhi** that takes in the correct input parameters to return the result of the expression

$$y(x, t) = \sin(kx - \omega t)$$

We will use this function to make a graph of y vs. x, and will step through time 't' to make an animation.

To set up the plots for animation, you will have to copy and paste the following code in after your **sinWaveZeroPhi** function:

```
# First set up the figure, the axes, and the subplots we want to animate
fig = plt.figure()
subplot = plt.axes(xlim=(0, 10), xlabel=("x"), ylim=(-2, 2), ylabel=("y"))
line1, = subplot.plot([], [], lw=2)
line2, = subplot.plot([], [], lw=2)
line3, = subplot.plot([], [], lw=2)

# Create a list of plot lines
lines = [line1, line2, line3]
```

This code sets up the proper size of the figure and subplots for the animation. This time ,we also label the x and y axes of the graph. Do not worry about the syntax of variables with the commas after them, since it is just there to make the code simpler to write and is not too important for this lab.

For Loops

One way to repeat specific lines of code is to use a **for loop**. We will use a for loop specifically to go through each element of a list in order. The following example illustrates a for loop going through a list of integers.

Example 4: *Suppose you had the list 'numbers' below. Print out each number.*

```
numbers = [1,5,3,6]
```

Solution:

```
for number in numbers:
     print number

# This will print:
1
5
3
6
```

In the loop above, we assign the variable 'number' (it can be any word, not just the plural version of the list name) to each integer in the list, and then run the body of the for loop. Every time the loop ends, we change the number to the next one until we run out of them in the list.
*Like a function, the body of a for loop **must** be indented.*

Methods

A **method** is a function that is part of a variable. Usually, methods are attached to the 'type' of a variable. A variable that is a string (a line of text) has a method called **capitalize** which capitalizes the first letter of the string. Some methods can also take input parameters, as in the following example.

Example 5: *Strings have a method called* ***split*** *that makes a list of words in the string separated by the the input parameter.*
Split the string called 'words' by the character 'r'
Solution:

```
words = "hirmyrnamerisralex"
words.split('r')

# This would return the list:
["hi", "my", "name", "is", "alex"]
```

In the code above, the variable is 'words', the variable's method is 'split', and the input parameter is "r".

You will now use loops and methods in your code to make a function that sets up the animation.

Lab Task: Define a new function called **init** that takes no input parameters.
This function will be called before the animation begins to set all of the plots to be empty before you begin adding data to them.
In the function body, write a for loop that goes through all of the elements in the list 'lines', and calls the method

```
set_data([], [])
```

on each element (there are two empty lists, `[]`, as the input parameters, representing the x and y lists, respectively). This sets the data for each plot line to nothing, and we will later change it to contain the appropriate function.

After the for loop, end the function by returning the 'lines' list by writing:

```
return lines
```

Note: Even though the 'lines' variable was not given as a parameter to the function* ***init**, as long as we have a 'lines' variable somewhere in the program (and we gave it a value earlier), then it will use that value.*

One last function to make!
Before you define it, we will go over one last concept.

Multi-dimensional lists

A **multi-dimensional list** is simply a list that contains lists.
In a regular 1-dimensional list like [1,2,3,4], you would get back the number 1 by accessing the list as so:

```
L = [1,2,3,4]
# The line below returns the integer '1'
L[0]
```

If you had a 2-dimensional list, you would first access the particular list you want, and then access an element in that list.
In the next example, we use a loop to access and change each element of the list.

Example 6: *Consider the list L below. Change each list in L to make its second element be 7.*

Note that we will use a ***different way*** *of making a for loop, by iterating through the indices of the list using the 'range' function.*
The 'range' function returns a list of numbers from 0 up to but not including the input parameter.
For example, `range(4)` *gives the list* `[0,1,2,3]`
We will use the 'len' function to get the length of the list.

Solution:

```
L = [[2,4],[3,8],[6,7]]
for i in range(len(L)):
    L[i][1] = 7

# This would make the list:
L = [[2,7],[3,7],[6,7]]
```

We can now make our last function.
This function will be the one that actually goes ahead and animates our graph. It will be called many times a second, and you will code it to properly update the plot figure each time it is called.

Lab Task:
After the init function, make an **independent variable** list called x using the 'linspace' function (it is similar to 'arange'. you can look up how it works online).

```
x = linspace(0, 10, 1000)
```

After the above line**,** Define a new function called **animate** that takes an input parameter called `i`.

We start by updating the function of the three plots.

In the function body, define a **dependent variable list** (like in Q1) called '`y1`' using 'x' as the independent variable list. y1 will be the list that evaluates your sinWaveZeroPhi function with amplitude 1, angular frequency 2π, and wave number $\frac{\pi}{2}$.
For the time, '`t`', we are going to use the input parameter '`i`', which always increases, allowing us to step through time. Set `t` to be $0.01 \cdot i$.
Hint: To get the value of π, use the variable 'pi', which was imported from numpy.

Next, make another dependent variable list '`y2`' with the same input parameters to sinWaveZero as above, but make the angular frequency -2π .

Because the value of 'i' increases at each call of the function, and the frequency is equal and opposite, the graph will model two propagating waves going at the same speed in opposite directions..

Make a third dependent variable list 'y3' that is of the sum of y1 and y2.
Reminder: You can get the sum of two lists **a** and **b** element-by-element by simply writing `a + b`

Still in the function body, put the independent variable list and dependent variable lists together as so:

```
waveFunctions = [[x,y1],[x,y2],[x,y3]]
```

Although all of the 'x' variables are the same, we will use this list to practise accessing 2-dimensional lists.

Write a for loop (in the animate function) that goes through the 'lines' list, and uses the `set_data` method for each line to set the appropriate function from the 'waveFunctions' variable. Use the **range** function and the **length of the list** like in Example 6 to do this instead of going through each element like you did earlier.

Here is an example of setting the first line to be the first function in waveFunctions. Use it to make a similar for loop.

```
lines[0].set_data(waveFunctions[0][0], waveFunctions[0][1])
```

After the for loop, end the 'animate' function by returning the 'lines' list by writing:

```
return lines
```

If you completed the last task, you have all of the functions needed to animate your graph. Now you have to call the animator and show the plot:

Lab Task: At the top of your python file, under the import statements you already have, add the line:

```
from matplotlib import animation
```

This imports the 'animation' function.
Now simply copy the following lines to the end of your file after your last function. The

```
anim = animation.FuncAnimation(fig, animate, init_func=init,
frames=200, interval=20, blit=True)

plt.show()
```

Run your code. You should see an animation of a standing wave pattern. Make sure to save your code as `standingLASTNAME.py`

Submit your code for both Q1 and Q2 to the email provided. Make sure you comment your code appropriately, and include your names and tutorial group at the top of your code files.