Waves Animations Module Student Guide

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, methods, external python libraries and 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:

$$y(x,t) = A\sin(kx - \omega t + \phi_0)$$

where A is the amplitude, k is the wave number, ω is the frequency, and ϕ_0 is the phase constant. The equation above is a function of position and time. [See Knight Section 16.3.]
- **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. [See Knight Section 17.8.]
- **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**. [See Knight Section 17.2.]

Lab Instructions

The Facilitator in your group should make a new shared Google Doc in which you will type out code. This code can be copied and pasted into Spyder where you can run it to test it. Some hints:

- Name the Google doc filename.py , where the py stands for python
- Choose "`Courier New`" as the font, as this is a fixed-width font where every character has the same spacing. That's good for visualizing code.
- When you indent a line after a `while` or a `def` function, use four spaces typed using a space-bar. Do not use the Tab button as you would in a code editor, as it won't copy and paste well into Spyder.
- Get a shareable link for your Google doc in which you allow anyone with the link to edit, and share this with your breakout room partners and your TA.
- The parts marked *Example* are meant to teach you python, but are not part of the final code. In order to show that you tried them out, please keep the examples in your final code, but comment them out.

Be sure to 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 is trying to do. The first comment at the top should list the names of all the partners in your group.

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

Activity 1: Beats History Graph (40 minutes)

Importing Libraries and Modules

Open up a new python file, and name it something unique, with the word "beats" in it. Do all the work for Activity 1 in this file. Be sure to make a shared Google Doc with the `Courier New` font, and indents as four-spaces, so that you can copy and paste into Spyder and share code with your breakout room teammates and your TA.

In order to model a wave function, we will be using python functions from external libraries. An external library is a collection of functions, written in python by somebody else, so that you don't have to write it yourself. Libraries are sometimes split into 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.

Programming Task A

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

```
from numpy import *
```

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

> (Note that, according to the Python Style Guide at https://www.python.org/dev/peps/pep-0008/ , "Wildcard imports (from <module> import *) should be avoided, as they make it unclear which names are present in the namespace, confusing both readers and many automated tools." A better way is to only import the functions that you are actually planning to use in the code. But, since we are at the start of this code, it's a little hard to know that. If, for style purposes, you really want to avoid the wildcard for this Practical, you can use `from numpy import sin, cos, pi, arange, linspace` , since these are the only five functions from numpy that you are probably going to use this week.)

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 should be commented out of your final 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, and show you get 3.*

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****. If you are using Google Docs to type your shareable code, I recommend indenting by typing four spaces.*

You should, of course, get the number 3 returned to you (the print function writes the contents of the variable answer to the python console).

At the end of Example 1, you should now go through and comment-out your addTwo definition and the test of it.

You will now create a function in your own code.

Programming Task B

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 $x = 0$, with a phase constant $\phi_0 = 0$ (hence the 'Zero' in the function name), at any time t. In Section 16.2 Knight calls a plot of such a function the "history graph" of the wave.

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 from the numpy library.

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. Once you are done testing the function, comment out your test-lines so they aren't part of the final code.

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 library. You will do it a little differently this time.

Programming Task C

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

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

pyplot is a module within the library matplotlib. This imports the pyplot module plotting functions into a variable you named 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. `arange` creates an array with evenly spaced values and returns the reference to it. The way it is called is `arange(start, stop, step)` , where `start` is the number (integer or decimal) that defines the first value in the array, `stop` is the number that defines the end of the array and isn't included in the array, and `step` is the number that defines the spacing (difference) between each two consecutive values in the array. Note that step can't be zero or you'll get an error.

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 a list `t` *of numbers from 0 up to and not including 7, with a step difference of 0.5 between each number. Then make a list* `y` *of the result of* `cos(t)`*. Check that individual elements of this list can be found with* `y[i]`*, where* `i` *is an index number that starts with* `i=0` *for the first element.*

Solution:

```
dt = 0.5
t = arange(0, 2, dt)
y = cos(t)
print(t[0], y[0])
print(t[6], y[6])
print(t[13], y[13])
```

At the end of Example 2, you should now go through and comment-out these lines.

It's good practice to create a variable `dt` for the step difference. You can now create one for your actual code.

Programming Task D

Create an independent variable list `t`, and two dependent variable lists of slightly different frequencies that call your `sineWaveZero` function on `t`.

Make a 0.01 step difference for `t`, and make the list from 0 up to and not including 5. In Python these are just dimensionless decimal numbers, but keep in mind your convention that time is measured in seconds, so this should produce time-steps of hundredths of a second over five seconds.

For both dependent variable lists, make sure the amplitude is the same. Each list represents the sine-wave sounds that you will be adding to create beats. Make the frequencies somewhere between 40-60 rad/s, with a difference of about 2-4 rad/s 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), like the one from Example 2, and show the plot.

Solution:

Let's use the same code from Example 2 again to plot cos for the first 7 radians:

```
dt=0.5
t = arange(0, 7, dt)
y = cos(t)
```

*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. Note that between fig.add and subplot there is an ***underscore****, which is shift-dash on my keyboard.*

```
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()
```

By default on my Spyder, the plot shous up in an internal "Plots" tab in the upper-right part of the Spyder window. In order to make a separate window that is easier to resize and screen-capture, you can type the following command into the Console of your Spyder window: `%matplotlib qt`.

At the end of Example 3, you should now go through and comment-out these lines.

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

Programming Task E

Create a plot figure and make three plots.

The first two plots are of the two `sineWaveZero` functions you created with slightly different freqeuncies.

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 keep a copy of your working python code in your shared Google Doc. Your TA should be able to copy and paste from this document into their own Spyder to check that your code actually works.

Activity 2: Standing Waves Animation (70 Minutes)

Open up a new python file, and name it something unique, with the word “standingwave” in it. Do all the work for Activity 2 in this file. Once again, make a shared Google Doc with the Courier New font, and indents as four-spaces, so that you can copy and paste into Spyder and share code with your breakout room teammates and your TA.

Copy the import statements from Activity 1 function into your new file.

Programming Task A

Define a function called `sineWaveZeroPhi` that takes in the correct input parameters to return the result of the expression

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

We will use this function to make a graph of y vs. x, and will step through time t to make an animation.

Next you will have to set up the plots for animation. These steps include a lot of matplotlib-specific lines that you may just copy-and-paste without fully understanding what they do. But you can always try adjusting things and re-running the code to see what changes do.

Programming Task B

Set up the figure, the axes, and the subplots we want to animate, by entering the following lines:

```
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)
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 1

Suppose you had the list 'numbers' below. Print out each number.

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

Solution:

```
for number in numbers:
    print(number)
```

At the end of Example 1, you should now go through and comment-out these lines.

In the loop in Example 1, 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. To make the Google Doc cut-and-pasteable with the Spyder window, you should use four spaces to indent.

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 2

*Strings have a method called **split** that makes a list of words in the string separated by the the input parameter.*

Split a string called `words` *by the character* `r`*. Create a list called* `wordbroken` *which contains the individual words between the rs as elements.*

Solution:

```
words = "hirmyrnamerisrannie"
print(words)
wordsbroken=words.split('r')
print(wordsbroken[0])
print(wordsbroken[1])
print(wordsbroken[2])
print(wordsbroken[3])
print(wordsbroken[4])
```

*In the code above, the **variable** is* `words`*, the variable's **method** is* `split`*, and the input **parameter** is* `r`*. At the end of Example 2, you should now go through and comment-out these lines.*

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

Programming Task C

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. Remember that if you are indenting twice, type 8 spaces (ie for a for-loop within a definition).

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

```
return lines
```

Note that 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]
print(L[0])
```

This would print the integer 1.

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 3

Start my making a list of lists, `L = [[2,4],[3,8],[6,7]]`*. 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]]
print ("Old 1st list: ", L[0][0], L[0][1])
print ("Old 2nd list: ", L[1][0], L[1][1])
print ("Old 3rd list: ", L[2][0], L[2][1])
for i in range(len(L)):
    L[i][1]=7
print ("New 1st list: ", L[0][0], L[0][1])
print ("New 2nd list: ", L[1][0], L[1][1])
print ("New 3rd list: ", L[2][0], L[2][1])
```

At the end of Example 3, you should now go through and comment-out these lines.

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 per second, and you will code it to properly update the plot figure each time it is called.

Programming Task D

After the `init` definition, make an **independent variable** list called `x` using numpy's `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 Activity 1) called `y1` using `x` as the independent variable list. `y1` will be the list that evaluates your `sineWaveZeroPhi` function with amplitude 1, angular frequency 2π, and wave number $\pi/2$. 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*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 `sineWaveZeroPhi` 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 practice 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 3 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.

Programming Task E

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.

```
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. I have found that this does not work in the Plot tab of the Spyder window. In order to make a separate window that actually makes the animation, you can type the following command into the Console of your Spyder window: `%matplotlib qt`.

Make sure to keep a copy of your working python code in your shared Google Doc. Your TA should be able to copy and paste from this document into their own Spyder to check that your code actually works.

⚠ **NOTE:** If your animation is not working, and you are running this from your laptop, it may be because the default settings in your Anaconda are not correct. On Spyder, select: Tools → Preferences → IPython console → Graphics → Graphics backend → Backend: select "Automatic" → click "Apply"

This Module was written by Mark Kazakevich, Mar.2015. Last updated for at-home Practicals by Jason Harlow, Mar.2021.