Physics 465: Computing Project 2

August 15, 2012

Representing Traveling Waves Visually with VPython

Last time we setup up a 3D arrow object that represented a phasor. This time, we'll create a 3D representation of a wavefunction of a free particle moving in the +x direction. Based on what we just learned we expect the wavefunction to be something like:

$$\Psi(x,t) = Ae^{i(kx-\omega t)} \tag{1}$$

where A is the magnitude of the wavefunction, k is the wavenumber ($2\pi/\lambda$) and ω is the angular frequency ($2\pi/T = E/\hbar$).

1) Some more python tricks

To complete this project we're going to need a couple more bits of python knowledge. These are introduced below.

1.1) Arrays

An array is a bit like a list, except that all the elements of an array need to have the same type (e.g., all integers, all floating point, etc.). Arrays have a lot of capabilities that lists don't have but we'll learn these as we go along. For the moment the main thing you need to know is how to create them and how to use them in the simplest context.

The easiest way to create an array is with the 'array' constructor:

```
x = array([1,9,4,3,5])

print x
print repr(x)
print type(x)
```

If you type the above code into the python interpreted you'll see the following:

```
>>> from visual import *
>>> x = array([1,9,4,3,5])
>>> print x
[1 9 4 3 5]
>>> print repr(x)
array([1, 9, 4, 3, 5])
>>> print type(x)
<type 'numpy.ndarray'>
>>>
```

You can see that if you 'print x' you get a list like output, but 'repr(x)' gives you something that looks like a constructor and 'type(x)' tells you the actual python type of the object (in this case it's 'numpy.ndarray'). There are several helpful utility functions that are nice for creating arrays. One I love is `linspace`. With `linspace` you can easily create an array with specific begin and end points with a specific number of elements. Here's an example... say I want an array with 5 elements including numbers varying from $-\pi$ to π:

```
>>> x = linspace(-pi, pi, 5)
>>> print x
[-3.14159265 -1.57079633  0.          1.57079633  3.14159265]
```

Notice that `linspace` automatically calculated ($-pi$, $-\pi/2$, 0, $\pi/2$, and π). This works just as well with 100 or 1000 element arrays, so you can see it's quite handy. Probably the single most useful property of arrays is that they can participate in arithmetic and they usually do the right thing. So for example:

```
>>> x**2
array([ 9.8696044,  2.4674011,  0.       ,  2.4674011,  9.8696044])
>>> cos(x)
array([ -1.00000000e+00,   6.12323400e-17,   1.00000000e+00,
         6.12323400e-17,  -1.00000000e+00])
>>> sin(x)
array([ -1.22464680e-16,  -1.00000000e+00,   0.00000000e+00,
         1.00000000e+00,   1.22464680e-16])
>>> exp(1j*x)
array([ -1.00000000e+00 -1.22460635e-16j,
         6.12303177e-17 -1.00000000e+00j,
         1.00000000e+00 +0.00000000e+00j,
         6.12303177e-17 +1.00000000e+00j,  -1.00000000e+00 +1.22460635e-16j])
```

When you pass an array in to a function, the function operates on each element of the array and returns a new array with the result of each operation in the corresponding element of the resulting array. Neat!

1.2) New orientation...

We want to represent the wavefunction of a particle moving in the x direction. At each value of x the wavefunction has a complex value. We can't really use the x direction to represent the real part of the complex number since that's the direction in which the complex number varies! So... luckily we're living in a universe with 3 large dimensions of space (we can discuss the small dimensions some other time). We can use one dimension for the motion of the particle, and have two dimensions left over for the real and imaginary parts of the wave function! This means we're going to switch from using the x and y components of our arrows as the real and imaginary parts to having the y and z components being the real and imaginary parts. So... the `SetArrowFromCN` function needs to change:

```
def SetArrowFromCN( cn, a):
    """
    SetArrowFromCN takes a complex number 'cn'  and an arrow object 'a'.
    It sets the  y  and  z  components of the arrow s axis to the real
    and imaginary parts of the given complex number.

    Just like Computing Project 1, except y and z for real/imag.
    """
    a.axis.y = cn.real
    a.axis.z = cn.imag
```

Notice that the only real change is $(x, y) \rightarrow (y, z)$.

1.3) The `range` function

There is a great list constructor called `range`. It was included in the doc for the last project, but it's quite simple. It just returns a list based on the arguments you pass. If you pass the length of a list as the only argument, it returns a list of indices for that list. So...

```
>>> for i in range(len(x)):
        print i, x[i]

0 -3.14159265359
1 -1.57079632679
2 0.0
3 1.57079632679
4 3.14159265359
```

1.4) Building a list of arrow objects

So.. now we have arrays and we know how to map complex numbers onto our arrows we can begin with the actual project. First let's create an array to keep track of the physical position of each or our arrows. Let's imagine we're looking at a portion of the x axis where the particle is expected to be with more or less uniform probability.

```
>>> L=6.0
>>> x = linspace(-L/2, L/2, 20)
>>> x
array([-3.        , -2.68421053, -2.36842105, -2.05263158, -1.73684211,
       -1.42105263, -1.10526316, -0.78947368, -0.47368421, -0.15789474,
        0.15789474,  0.47368421,  0.78947368,  1.10526316,  1.42105263,
        1.73684211,  2.05263158,  2.36842105,  2.68421053,  3.        ])
```

So the x array is just a set of 20 values from -3.0 to +3.0. Let's make an arrow at each of these positions using a simple loop.

```
for i in range(len(x)):
    a = arrow(pos=(x[i], 0, 0),     # on the y,z axis at location 'x'
              axis=(0,1,0),         # pointing in the 'real' direction
              color=color.red)      # make it red. ;->
    alist.append(a)                 # add to list
```

So now we have a list of 20 red arrows all pointing up!

1.5) Applying the wave function

How can we compute the value of the wave function (Eq. 1) at these positions? Easy! We just use the feature of arrays that let's us compute the value of a function for each element of the array. Let's start by getting the function at $t = 0$. Then the wave function is simply:

$$\Psi(x, 0) = Ae^{ikx} \tag{2}$$

Just to make sure we can see something, let's set k to $3\pi/L$ (that means in a distance L the phase will change by 3π, or one and a half cycles).

```
>>> k=3*pi/L
>>> psi=exp(1j*k*x)
>>> psi
array([ -1.83690953e-16+1.j        ,  -4.75947393e-01+0.87947375j,
        -8.37166478e-01+0.54694816j,  -9.96584493e-01+0.08257935j,
        -9.15773327e-01-0.40169542j,  -6.14212713e-01-0.78914051j,
        ... ])
```

```
>>>
```

Now we just have to use the modified `SetArrowFromCN` function in our construction loop to set the direction of each phasor set correctly. Here's the whole project so far:

```
from visual import *

L=6.0                           # range of x is 6 units
x = linspace(-L/2, L/2, 20)     # from -3 to +3
k = 3*pi/L                      # set up the wave number
psi = exp(1j*k*x)               # set up the initial wave function

alist = []                      # an empty list for our arrow objects

def SetArrowFromCN( cn, a):
    """
    SetArrowFromCN takes a complex number  cn  and an arrow object  a .
    It sets the  y  and  z  components of the arrow s axis to the real
    and imaginary parts of the given complex number.

    Just like Computing Project 1, except y and z for real/imag.
    """
    a.axis.y = cn.real
    a.axis.z = cn.imag

for i in range(len(x)):
    a = arrow(pos=(x[i], 0, 0),       # on the y,z axis at location 'x'
              axis=(0,1,0),         # pointing in the 'real' direction
              color=color.red)      # make it red. ;->
    alist.append(a)                 # add to list
    SetArrowFromCN( psi[i], a)      # set up arrow from wave function
```

If you run the program at this point you should see something that looks like Fig. 1 (expect that the background will probably be black).

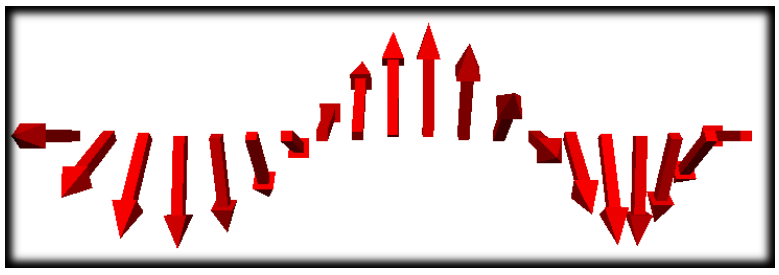

Figure 1: traveling wave

1.6) Turning on the time

Next we need to add the time component as in Eq. 1. Add a set of nested loops, similar to last weeks loop that sets the orientation of each arrow in the list based on the current time and the initial phase of the arrow at each position. Something like this:

```
omega = 2*pi                    # 1 rev/sec
t=0.0                           # start t at zero
dt=0.01                         # 1/100 of a second per step

while 1:
    rate(100)
    t+=dt
    for i in range(len(x)):
        #
        # Put your code here to set the orientation of the "i"th arrow
        # in the list "alist".
        #
```

Questions

Please answer these questions at the end of your report.

1) Which way does the wave appear to move? Why is it moving this way?

2) With what velocity do the wave "crests" move? Why?

3) What could you change about the *spatial* behavior of the wavefunction to make the waves appear to move in the opposite direction? No fair modifying the time part! Use your program to verify your answer. How does changing the spatial behavior of the wavefunction affect the expectation value of the momentum operator?