

Physics 465: Computing Project 1

August 9, 2012

1 Representing Complex Numbers Visually with VPython

One of the primary learning goals of this course is for you to develop an intuitive understanding of what it means to *do* quantum mechanics and how the machinery works. I have found that it helps many students to have a concrete visual image or *representation* of what's going on. Since quantum mechanical probability amplitudes are actually complex numbers, it's necessary to have a visual representation of a complex number.

1.1 Preliminaries

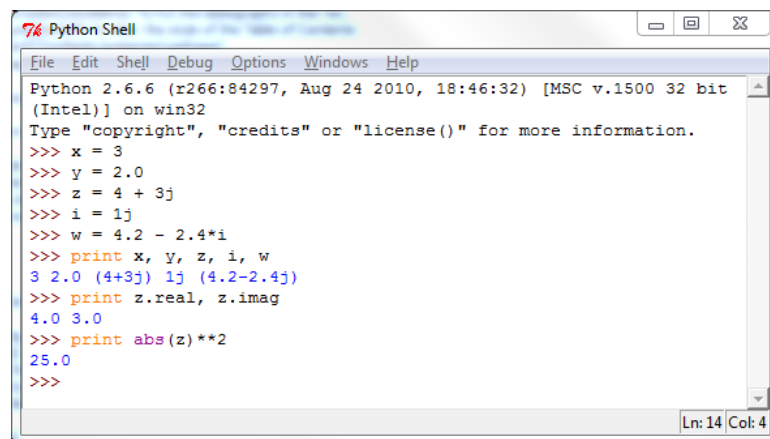
Since this is our first VPython project we'll need a few preliminaries to get started. First we need to learn a bit about python object types and some python syntax. For our purposes this week there are basically only a few important types: scalars, lists, and 3D objects. We'll discover some other types in later projects.

1.1.1 Scalars

Scalars are just regular old numbers.

```
x = 3
y = 2.0
z = 4 + 3j
i=1j
w = 4.2 - 2.4*i
print x, y, z, i, w
print z.real, z.imag
print abs(z)**2
```

If you run the Python Shell and enter these commands one at a time, you'll see something like figure 1 as a result.

A screenshot of a Python Shell window. The window has a menu bar with 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Windows', and 'Help'. The main text area shows the following code and output:

```
Python 2.6.6 (r266:84297, Aug 24 2010, 18:46:32) [MSC v.1500 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> x = 3
>>> y = 2.0
>>> z = 4 + 3j
>>> i = 1j
>>> w = 4.2 - 2.4*i
>>> print x, y, z, i, w
3 2.0 (4+3j) 1j (4.2-2.4j)
>>> print z.real, z.imag
4.0 3.0
>>> print abs(z)**2
25.0
>>>
```

The status bar at the bottom right indicates 'Ln: 14 Col: 4'.

Figure 1: How Some Scaler Operations appear when executed in the Python Shell

1.1.2 Lists (and Tuples)

Lists are a kind collection that have a definite order. An example of a list is:

```
x = [1,9,4,5]
```

Because the list has a certain order, you can reliably retrieve objects from the list based on their place in the list. The “list place” is called the “index”. Like the “c” language, python begins list indices at the value “0”. So, if you asked for the 2nd element of this list, you’d get the number “4”, not “9” as you might expect.

```
x = [1,9,4,5]
print "x[0] ->", x[0]
print "x[1] ->", x[1]
print "x[2] ->", x[2]
print "x[3] ->", x[3]
```

If you execute this code, you should see the output below. This seems a little strange at first, but you’ll find that it’s actually got several handy side effects that we’ll come to rely on in time.

```
x[0] -> 1
x[1] -> 9
x[2] -> 4
x[3] -> 5
```

You can add things to the end of a list with the “append” method and insert things into the beginning or middle of a list with the “insert” method. If you type the commands that follow into a python shell, you’ll get the output shown in figure 2.

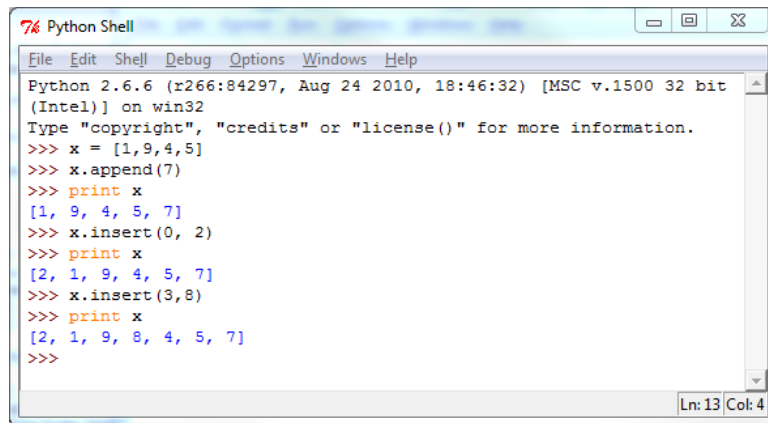


Figure 2: How Some List Operations appear when executed in the Python Shell

```
x = [1,9,4,5]
x.append(7)
print x
x.insert(0, 2)
print x
x.insert(3,8)
print x
```

A “Tuple” is just like a list except that it can’t be changed once it’s created, so it’s an “immutable” list. While you create lists using the “[]” (square bracket) delimiters, to make a tuple you just use parenthesis: “()”, like so:

```
x = (1,9,4,5)
```

Tuples, like lists, use “0” as the index of the first element of the tuple.

1.1.3 3D Objects

3D objects are geometrical objects that can be seen on the computer display, such as a “sphere”, “box”, “arrow” and so on. You will find documentation for these on the vpython web site, or in the “help” menu of your IDE (called VIdle). You create 3D objects by first importing the “visual” module like so:

```
from visual import *
```

For the purposes of this project the only object we’ll be using is the “arrow”. Remember that a complex number can be visualized as a “phasor” and a phasor is basically an arrow! Below you’ll see python code that creates a red arrow of length 1 pointing in the +y direction and a blue arrow of length 2 pointing in the +x direction.

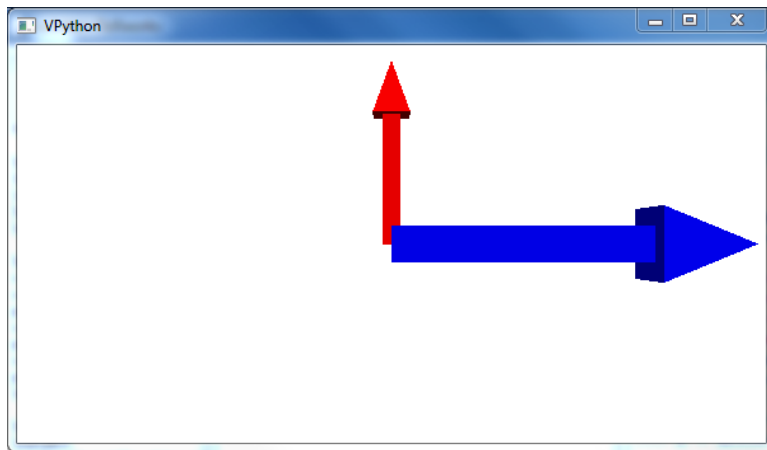


Figure 3: How to create two 3D arrows in the Python Shell

```
from visual import *  
a = arrow()  
a.axis = (0,1,0)  
a.color = color.red  
b = arrow(axis=(2,0,0), color=color.blue)
```

If you execute these commands in the python shell you should see something more or less like figure 3 (you may notice the background is black rather than white as shown here.)

1.2 Flow Control

In order to have the program repeat a procedure many times we'll need to learn some python "flow control". This just means the programs we write will need to have loops that depend on various conditions. The most basic loop in python is the "while" loop. Here is a simple example:

```
x = 1.0  
print "before loop, x is:", x  
while x < 5.0:  
    print x  
    x = x + 1.0  
  
print "loop finished."  
print "x is now:", x
```

When you execute this in a shell, you should see:

```

before loop, x is: 1.0
1.0
2.0
3.0
4.0
loop finished.
x is now: 5.0

```

Another common loop is the “for” loop. In this case python will iterate through the contents of a list and execute a series of statements for every element of a list. This can be quite handy if you already have a list you want to iterate through. The “range” function is a nice way to quickly generate lists. For example “range(4)” generates the list [0,1,2,3], the indices of a list of length 4! Based on that idea, see if you can figure out how this code works:

```

x = [1,9,4,5]
for i in range(4):
    print "x[" ,i, "] -> ", x[i]

```

Which produces the output:

```

x[ 0 ] -> 1
x[ 1 ] -> 9
x[ 2 ] -> 4
x[ 3 ] -> 5

```

One last point.. to keep things changing at a reasonable rate there is a function you can call to limit the frame rate in VPython. If you want your loop to execute no more than 100 times per second, no matter how fast your computer, you can include “rate(100)” in your loop code. This will ensure that your program runs at a standard rate on all computers.

1.3 Functions

Finally we’ll want to use a function to do the busy work of making the real and imaginary parts of a complex number show up as the cartesian coordinates of the tip of our arrow. A function is just a bit of code that you need to run at different times. It’s nice to encapsulate that code in a function so you can use it easily whenever you need to perform some task. The function we need takes a complex number ‘cn’ and makes an arrow ‘a’ represent that complex number in 3D space by setting the cartesian coordinates of the tip of the arrow to the real and imaginary parts of the complex number. The syntax looks like this:

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

This seems like such a simple function that we might not even bother, but we'll see that it de-clutters other parts of our program where a little clutter can be very distracting. Notice the comment in triple quotes. These are very important. When you're writing your own functions it's always a good idea to embed comments that make your code easier to understand.

2 A Phasor getting multiplied by a small pure phase.

Let's say we start with a complex number like $z = 2e^{i\pi/4}$. What happens to that phasor if we repeatedly multiply by a complex number with magnitude of one, but with a small non-zero phase angle $c = e^{i\delta}$? Let's find out! First let's get a good representation of our original complex number. Type in the following code and run it. Make sure you understand each line. If you have any questions, please ask. When the program runs you should see a 3D representation of our starting complex number. Does it look reasonable? Is it pointing in the right direction?

```
from visual import *

i=1j                                # Let's use 'i' for "imaginary"
a = arrow(color=color.red)          # red is a nice color
z = 2.0*exp(i*pi/4.0)

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

SetArrowFromCN(z, a)
```

2.1 Write A Loop

Now.. time for you to write some code. Write a loop, similar to the one in the example, that repeatedly multiplies this complex number 'cn' by a pure phase $c = e^{i\delta}$ where *delta* is a small angle, like $\delta = 0.01$ or so. Each time through the loop 'cn' should be updated to its new value, just like 'x' had a new value in the example. Does the phasor appear to be doing something reasonable? Don't forget to include a 'rate' function so you can actually see the changes happening at a constant (and reasonable) rate.

2.2 Questions

Please answer these questions at the end of your report.

- 1) What happens to the magnitude of the complex number 'cn' during the execution of your loop?
- 2) What happens to the phase of the complex number 'cn' during the execution of your loop?
- 3) How many iterations of your loop does it take for the phasor representation to complete a full cycle? Can you explain why this is a reasonable amount of time?
- 4) What is the cartesian form of the number 'c'? Demonstrate analytically that multiplying by this number has the effect you observed on both the magnitude and the angle of the phasor?