Visualizing Electric Fields with VPython

Python is a popular and versatile programming language that we will use in this course to create simple simulations of electric and magnetic fields. VPython is a module that allows basic 3-D modelling in the language of Python. This lab manual will walk you through the creation of a program that simulates the field of an electric dipole.

Double-click on the desktop icon "VIDLE for VPython". This starts a modified version of the standard Python editor IDLE called VIDLE. In this window you can start entering the code for your program. We would like our program to do the following:

- Define an electric dipole—a pair of charges of equal and opposite magnitude.
- Define a grid of locations at which we will determine the strength and direction of the electric field created by the dipole.
- Calculate the electric field at each location on the grid and display it with an arrow.
- Place a test charge in the field and simulate its motion.

As programming experience is not a prerequisite for this course, the following assumes you are not familiar with programming, let alone VPython. More experienced users can move through the instructions more quickly.

Part A: Python basics

To begin, type

 # This program simulates the field generated by an electric dipole

The '#' sign indicates that this line is a "comment". That is, when the Python interpreter executes the program line by line, it will notice the # and proceed to ignore the whole rest of the line. Comments are there just to help a human make sense of what the code is doing. You should get in the habit of commenting your own code. Any programmer will tell you that there are few things worse than trying to make sense of someone else's code that has not been properly commented.

Press 'Enter' twice and type the following:

from visual import *

This comes at the beginning of every VPython program. It is telling the program to load all the definitions in the module **visual** that is part of the VPython package. These definitions include objects like spheres and arrows that we will use shortly.

Now hit “Enter” twice and add the following three lines of code to your program:

```
# Definition of important constants
kcoulomb = 9e9
qproton = 1.6e-19
```

The first line labels this as the section where we are defining important constants that the program will use. The second and third lines define Coulomb’s constant and the charge of a proton in SI units. Formally we are defining two variables named ‘kcoulomb’ and ‘qproton’ and using the assignment operator, ‘=’, to give them the scientifically notated values 9×10^{9} and 1.6×10^{-19}. Although these variable names could be shortened to just ‘k’ and ‘q’ without causing much confusion here, it’s a good idea to give descriptive variable names so that there is no chance of ambiguity. Variable names can only consist of letters, numbers, and underscores.

Note that Python doesn’t associate any physical units to the values of variables; it just treats them as pure numbers. You might consider adding a short comment (beginning with a ‘#’ symbol) on the same line after each variable definition just to remind yourself what physical units correspond to those values.

Now save your program by going to “File” and then “Save” and typing the filename “ElectricDipole.py”. The “.py” extension must be included so that the computer knows that what you are typing in the editor should be interpreted as a Python program and not just a text file.

To run the program you can go to “Run” and then “Run Module” or just hit the F5 key. A new window will pop up called the Python Shell. This is a command line interface that you can use interactively to modify programs as they are running. For now we will just be using it to show us the output of our program. Unfortunately the program didn’t return any output to the Shell. This is because we have not included any instructions to print the output on the screen.

On a new line add the following code to your program:

```
print qproton
```

Now run it to see what it does. The “print” command writes the value assigned to the variable ‘qproton’ to the Python Shell when the program is executed. As it stands, the output has no context. Replace the above line with the following one and run the program:

```
print “The charge of a proton is”, qproton, “Coulombs.”
```

This outputs the material in the quotes (called a string) exactly as it is written and inserts the value of the variable 'qproton' naturally into the output. If you accidentally forget the commas and try to run the program you will encounter a "syntax error", meaning that something in your program cannot be understood by the Python interpreter because it does not obey the rules of the language. Save your program.

Part B: Defining the electric dipole

Now skip a line or two (this whitespace is all ignored by Python, but it makes reading your program easier) and add the following:

```
# Definition of the charges
plus = sphere()
```

If you run your program now a window will pop up displaying a gray sphere. We have defined a new variable named 'plus' and have assigned it to an object now instead of a number. The sphere object is defined in the module that we loaded, **visual**. You can play around with the camera using the mouse. Scrolling while holding both the left and right mouse buttons (or use Alt+LeftClick) will zoom in and out and scrolling while holding just the right mouse button (or Ctrl+LeftClick) will rotate the camera about the origin (in this case the center of the sphere. Note that while the graphics window is open your program is still running. To end the program just close the graphics window.
The sphere that appeared is the default sphere when you do not specify any arguments within the parentheses. Modify the sphere by replacing the last line with the following:

```
plus = sphere(pos = vector(-2e-10, 0, 0), radius = 1e-11, color = color.red)
```

When you run the program you should now see a smaller red sphere on the left side of your screen. The three arguments define the x-y-z position of the sphere, its radius, and its color. All three of these are predefined attributes of the sphere-type object defined in **visual**. Notice that the position is a vector attribute whereas the radius is a scalar attribute. Python will rightfully complain if you try to feed it nonsense by treating position as a scalar and radius as a vector. The color attribute references a predefined value 'red' here, but you can assign arbitrary colors by using 'color = vector(R, G, B)'. Here R, G, and B are the amounts of red, green, and blue to mix together, each measured on a scale between 0 and 1. Just replace each letter with the value you want—for example, 'color = vector(1, 0, 0)' would be pure red. The default sphere (without any arguments, that is) has position = vector(0,0,0), radius = 1, and color = vector(1,1,1) (which is white). The reason why the new sphere only looks slightly smaller than the old one is that the camera automatically chooses an appropriate initial zoom value.

Add an additional attribute to your sphere after 'color' and call it 'q'. Assign it a value of 'qproton'. The attribute 'q' is not associated any visual property of the sphere object, nor is it predefined in **visual**. Instead it is just an extra number associated to the object 'plus' that will be used as the sphere's charge to calculate the electric field in the region nearby.

Now on a new line create another sphere object called 'minus'. Position it at an x-coordinate of +2e-10 with the same radius as 'plus'. Make it blue and assign it a charge of '-qproton'. Save your program.

Part C: Calculating the electric field

Before we can calculate the electric field E at a point we must define a variable that provides the locations at which we are interested in finding the field. Let's start with a single point right between the two charges. Add the following to your program:

```
# Definition of the grid of points at which to calculate the field
locations = vector(0, 0, 0)
```

We'll come back and add more points in Part D, but for now it's easiest to calculate the electric field at just one point. Add another two lines of code:

```
# Routine to calculate the electric field
E = vector(0, 0, 0)
```

This second line initializes the electric field variable 'E' as a vector of zero length for now.

Now we need to find the relative position vector $\vec{r}$ between the charge and the location at which we want to find E. This is just the difference of the vectors representing the location we're interested in and the position of the source. Write a line of code given by

r = ***{****Insert your own expression here****}***

to define the position vector from the positive charge to the location we want to evaluate E. The variable 'plus.pos' is the vector that gives the position of the positive charge. (Note: The variables for the actual components of the charge's position vector are 'plus.pos.x', 'plus.pos.y', and 'plus.pos.z'. You do not need to use these if you are just interested in the sum or difference of two vectors. Python will automatically add or subtract two vector-type variables component by component.)

Add a line of code outputting the position vector:

print "The relative position vector is", r

Now that you have the position vector, you will need to find its magnitude 'rmag' and the unit vector 'rhat'. The magnitude of $\vec{r}$ is given by $|\vec{r}| = \sqrt{r_x^2 + r_y^2 + r_z^2}$. Translate this into a new line of code

rmag = *{Insert your own expression here}*

using the function 'sqrt()' and squaring with the '**' (double star) operator (e.g., $\sqrt{3}$ can be expressed as 'sqrt(3)' and r_x^2, the square of the x-component of r, can be computed by the expression 'r.x**2'). Add this line of code to print 'rmag':

print "The magnitude of the relative position vector is", rmag

Once you have inserted the line for 'rmag', define the unit vector 'rhat' using a new line of code:

rhat = *{Insert your own expression here}*

and then add a line that prints out 'rhat' just as above.

With the relative position vector to the positive charge now calculated (along with its magnitude and unit vector), draw an arrow to display it on the screen:

pos_arrow = arrow(pos = plus.pos, axis = r, color = color.white, shaftwidth = 1e-11)

For this new arrow-type object 'pos_arrow', the position attribute is the vector locating the arrow's tail and the axis is the vector providing the arrow's orientation. You should see a white arrow representing the relative position vector show up when you run your program. Afterwards, comment out the line defining 'pos_arrow' using a # sign at the beginning of it.

Finally we can calculate the value of the electric field variable 'E' at the location of interest due to the positive charge. Add a line of code that looks like

E = E + *{Insert your own expression here}*

where inside the braces you should add a calculation using the formula for the electric field at a point due to a point charge. This expression should involve 'kcoulomb', 'plus.q', 'rmag', and 'rhat'. Note that the above line does not appear to make sense as a mathematical equation. This is okay because the = operator does not mean mathematical equality in Python, it just means

assignment. Here we are evaluating the right-hand side first by adding the old value of the vector 'E' (which was (0, 0, 0)) to the calculated vector of the electric field due to the positive charge and storing it back in the same variable 'E'. You can add a line at this point outputting 'E':

print "The value of the electric field due to the positive charge is", E

Now that we've calculated 'E', we can create a new arrow to visualize it. Add the following:

E_arrow = arrow(pos = locations, axis = E_scale * E, color = color.yellow)

Here we have defined 'E_arrow' to have a tail located at the vector given by 'locations' and an orientation in the direction of 'E'. The quantity 'E_scale' is an arbitrary scale factor to make the arrow comparable in size to the two spheres (otherwise the length of the arrow would be enormous because the value of 'E' is much larger as a number than the radius of the spheres). Define a new variable 'E_scale' right beneath 'qproton' in your section of important constants and give it a value of 4e-22.

All of this has only calculated the electric field due to the positive charge. We could just copy-paste our calculation code again below and make a few modifications to also include the negative charge, but a far more effective solution is to use a loop. Add a new object called 'dipole' beneath the definitions of 'plus' and 'minus' as follows:

dipole = [plus, minus]

This new object is called a list. The brackets define what items are bundled together into the list, in this case the two sphere objects 'plus' and 'minus'. With this list we can modify what we had previously written and place the calculations we just did inside a loop over both charges. Modify the section 'Routine to calculate the electric field' as follows:

 # Routine to calculate the electric field
E = vector(0, 0, 0)
for charge in dipole:

{Include here all your previous calculations of 'r', 'rmag', and 'rhat' for a charge as well as the calculation for the electric field it generates. Make sure it's indented relative to the 'for' line above. Change any use of the word 'plus' here to the word 'charge' so that it references the respective element of 'dipole', and not just 'plus'}

E_arrow = arrow(pos = locations, axis = E_scale * E, color = color.yellow)

Here we have a ‘for’ loop. In this type of loop each iteration steps through the items in a list (in this case each item ‘charge’ in the list ‘dipole’) and all of the code that is indented after the colon is executed for each item in the list. The program will apply the indented code first to the object ‘plus’, which is first item in the list, and then the object ‘minus’.

Notice that ‘r’, ‘rmag’, and ‘rhat’ are redefined when stepping to the next ‘charge’ in the list, but ‘E’ just gets added to and does not get reset. So the value of ‘E’ after the loop is done is the total value of the electric field at the point in ‘locations’ due to all the charges (in this case there are just two, but you could easily define more spheres and add them to the list). Note that after the loop is done the arrow for the electric field at that point is now plotted, this time using the total value of ‘E’ from all charges. Make sure you run your program to see that your output makes sense and then save it.

Part D: Defining the grid of points

Now that we’ve established how to calculate the electric field at one point, generalizing this to a calculation at many different points is easy. We simply have to create another loop over each of the elements that we are going to add to ‘locations’. Take all of the code that we just edited in the above section and put it within another loop:

```
# Routine to calculate electric field
for point in locations:
    E = vector(0, 0, 0)
    {Put the ‘for’ loop to calculate the electric field resulting from both charges here.
    Make sure you change any instance of ‘locations’ in the loop to ‘point’, so that the
    loop now uses the respective element of ‘locations’ appropriate for that iteration.}
```

This guarantees that every point in locations will be assigned an ‘E_arrow’. Now we just need to add more points to ‘locations’. We could make ‘locations’ a list of vectors that we manually add one by one, but this is extremely tedious and counterproductive. A better solution is to define a grid of vectors and append them to ‘locations’. Replace the section that previously defined ‘locations’ with the following code:

```
# Definition of the grid of points at which to calculate the field
locations = []
dx = 1e-10
dy = 1e-10
dz = 1e-10

for x in arange(-4.5e-10, 4.5e-10 + dx, dx):
```

```
    for y in arange(-4.5e-10, 4.5e-10 + dy, dy):
        for z in arange(-4.5e-10, 4.5e-10 + dz, dz):
            a = vector(x, y, z)
            locations.append(a)
```

Let's break this code down. We've now defined 'locations' as an empty list, along with increments 'dx', 'dy', and 'dz'. Then we define three 'for' loops nested within each other. Each loop spans a discrete set of coordinates over 'x', 'y', and 'z', each starting at -4.5e-10 and spanning up to 4.5e-10 in increments of 1e-10. (Note that the form of 'arange()' is 'arange(min, max, increment)', and it's a function that generates a list of equally-spaced values up to but not including its second argument. So the lattice goes up to 4.5e-10, but not 4.5e-10 + 1e-10 = 5.5e-10). Inside the loops, each set of coordinates is assigned to a temporary vector 'a' and 'locations.append(a)' appends the current vector 'a' to the list 'locations'. Now 'locations' will contain the vectors to each point of a cubic lattice within the specified range. As a side note, the reason for choosing coordinates with half-point values is because a point located exactly on one of the charges would cause a 'division by zero' error in the calculation of the electric field there.

Before running your program, comment out each line within the 'for' loops of the electric field calculation that prints output to the Shell. Printing to the screen is so slow that it will take your program forever to finish when it does this for each of ~1000 points. The program should now show a grid of arrows representing the electric field. Save it once you are satisfied with the output.

Part E: Simulating the motion of a proton in the dipole field

Let's simulate the dynamics of a proton moving in this electric field. Define a new object on a line right beneath the definitions of 'plus' and 'minus' as follows:

```
    proton = sphere(pos = vector(5e-10, 3e-10, 4e-10), radius = 1e-11, color = color.cyan,
q = 1.6e-19, m = 1.7e-27, p = vector(0,0,0), trail = curve(color = color.white))
```

All of the above code should be on the same line. Here the new variables are the proton mass 'm' (given in kg), the proton momentum 'p', and a trail to follow the motion of the proton from step to step. Now add the following to the end of your program:

```
# Simulate the trajectory of a proton
dt = 1e-17
t=0
while t < 3e-13:
    rate(1000)
    t = t + dt
```

```
E = vector(0,0,0)
for charge in dipole:
        r = proton.pos - charge.pos
        E = E + kcoulomb * charge.q * norm(r) / mag(r)**2
F = proton.q * E
proton.p = proton.p + F * dt
proton.pos = proton.pos + (proton.p / proton.m) * dt
proton.trail.append(proton.pos)
```

Here ‘while’ is another type of loop that runs until the given condition (here ‘t < 3e-13’) is met, norm(r) and mag(r) are predefined functions that allow you to quickly find ‘rmag’ and ‘rhat’ that we calculated earlier, the ‘rate()’ command updates the display window, and the ‘proton.trail.append()’ command appends the latest step to the overall trail tracking the proton’s trajectory. From your knowledge of physics, can you figure out what each of the other steps are doing?

Experiment with changing various parameters in your program to see how the behavior of the proton changes.

Lastly, add a little something extra to your program—your own creative contribution to set it apart from what everyone else has done so far.