Calculating and displaying the electric field of a uniformly charged ring

1 Problem Statement

Write a VPython program which calculates the electric field of a uniformly charged ring at various locations specified in this handout, including locations not on the axis of the ring.

The ring is made of thin glass of thickness 1 mm.
The radius of the ring is $R = 3$ cm and it lies in the yz plane with its center at the origin, axis on the x axis.
The ring is rubbed with silk and acquires a total charge $Q = 50\,\text{nC} = 50 \times 10^{-9}\ \text{C}$.
We will make the approximation that the charge is uniformly distributed around the ring.

On a grid of observation locations near the ring, you will display an arrow representing the net electric field at that location, due to all the charges on the ring. By writing a program you can determine the electric field at *any* location in space, not just locations along the axis of the ring.

2 Diagram and planning

On paper or a whiteboard, make a rough 3D sketch of the situation.

- Divide the ring into many short slices.
- Show a typical short slice of the ring at some location R*< 0, cos(theta), sin(theta) >, where theta is an angle measured from the y axis going counterclockwise around the ring, as viewed from a position along the +x axis, looking toward the origin.
- Show and label the position vector from that slice to the observation location < 1, 2, 0 > cm.
- Show and label the electric field vector at < 1, 2, 0 > cm due to this slice .

Your task is to tell the computer how to calculate the contribution of one slice, and to add up all the contributions of all the slices.

**Make sure your group agrees on their understanding of the geometry of the situation.
Then discuss your diagram with a neighboring group.**

3 Write the program

Your program should be organized with these sections:

```
## setup and constants
## objects
## initial values
## calculations
```

3.1 Setup and constants

Start with these two statements:

```
from  __future__  import division
from visual import *
```

In the first line note the two underscores before and two after the word “future”.
This make Python treat (1/2) as a floating point number (0.5) instead of an integer (0).

After typing the first two lines to invoke VPython, enter the values of the physical constants you will need. You will also eventually need a scale factor for representing electric field vectors with arrows, so go ahead and create it now, temporarily giving it a value of 1.0 (you will change this later).

```
## constants
oofpez = 9e9 # One Over Four Pi Epsilon-Zero
scalefactor = 1.0 # temporary value
Q = 50e-9 # total charge
N = 20 # cut up the ring into this many slices
R = 0.03 # radius of ring (thickness of ring is 0.001)
dx = 0.005 # used in constructing a grid of observation locations
dy = 0.02 # used in constructing a grid of observation locations
```

3.2 Objects

In your program add this comment line:

```
## objects
```

Display a cyan ring (`color=color.cyan`) in the yz plane, with its axis along the x axis, with radius R, 1 mm thick. See the Visual reference manual available on the Help menu.

3.3 Initial values

To get started, you will calculate the net electric field at one observation location. Later, you will modify this, and write a loop to calculate and display the electric field at many observation locations. You will need to have a variable that contains the value of the observation location. In your program add this:

```
## initial values
obslocation = vector(dx,0,0) # dx is a previously defined constant
```

3.4 Calculations

You can't use the "analytical" formula derived in the textbook for the electric field of a ring in these VPython calculations, because you are going to calculate the field at many locations not on the axis, where the formula doesn't apply. You must go back to the superposition principle. At the observation location, you must calculate the electric field due to each slice of the ring, and add up the many contributions. The reason why we calculate the field at < dx, 0, 0 > first, on the axis, is that we can use the analytical formula to check our calculation.

VERY IMPORTANT

Refer to your diagram to see what quantities you need to tell the computer to calculate.

- Calculate the net electric field at the location given by "obslocation". For each of the N slices you need to calculate

    a vector from the source charge (one slice) to the observation location,
    the magnitude of that vector, and
    the corresponding unit vector,

in order to be able to calculate the electric field vector contributed by that slice. You need to add the contributions of all the slices to get the net field.

The magnitude of a vector `r` can be calculated as `sqrt(r.x**2 + r.y**2 + r.z**2)`, or more easily as `mag(r)`.

When you calculate the electric field of a dipole, there are only two source charges (+q and –q). But for the ring, there are N slices, where N could be a large number of slices, so even to find the net electric field at one observation location you have to write a loop that runs through all N slices.

The organization of your program should look like this:

```
from  __future__  import division
from visual import *
## constants
... [your constants here, including scalefactor]
## objects
... [your ring here]
## initial values
[your initial values from above]
obslocation = vector(dx, 0, 0)
## calculations
theta = 0
Enet = vector(0,0,0) # for adding up the individual contributions
while theta < 2*pi:
   ## Calculate new vector value for source location, using theta
   sourceloc = R*vector(0, cos(theta), sin(theta))
   ## Calculate dE (a vector) at obslocation due to this one slice.
```

```
    ## How much charge is on this one slice? Total charge Q, N slices...
    [your code here to calculate dE, due to this one slice]
    ## increment Enet to add up all the contributions
    Enet = Enet + dE
    ## calculate new value of theta in radians, to move to next slice
    theta = theta + 2*pi/N
print obslocation, Enet ## when loop is finished, print location and Enet
```

- To check your calculation, compare your answer to the answer you get using the formula for the electric field on the axis of a uniformly charged ring (use VPython to evaluate and print this result). The analytical formula for the magnitude of the electric field on the *z* axis of a uniformly charged ring in the xy plane is

$$E_{\text{ring}} = \frac{1}{4\pi\varepsilon_0}\frac{qz}{\left(R^2+z^2\right)^{3/2}}$$

Make sure the result of your numerical integration agrees with the analytical formula.

3.5 Arrow representing electric field at one location

Given the size of the display (in meters), a reasonable length for an arrow representing the electric field vector would be something like 0.01 m, so you need to scale the electric field to this approximate length. You want $|\vec{E}|*\texttt{scalefactor}$ to be about 0.01, so write $\texttt{scalefactor} = 0.01/(\text{typical E value})$.

You already printed a typical E value near the ring, so use that value to calculate `scalefactor` in the constants section of your program. You may need to adjust `scalefactor` later.

- Just after your print statement, create an orange arrow representing the electric field at the observation location, and scale it appropriately by setting the arrow's axis to the scalefactor times the electric field. The ring and the arrow must be clearly visible. If you don't remember how to create an arrow, check the Visual reference manual available on the Help menu. Note that the axis of the arrow is relative to the tail of the arrow.

Make sure everyone in the group agrees that the display looks reasonable.
Then check your display with a neighboring group.

4 Adding a grid of observation locations

To see the pattern of electric field around the ring, you will extend your program to calculate the electric field on a grid of observation locations near the ring.

Instead of copying and pasting code many times, you'll put your existing calculations inside x and y loops. To be inside these two loops, you need to doubly indent everything needed to calculate and display the electric field at one location, as listed here:

```
## calculations
theta = 0
Enet = vector(0,0,0) # for adding up the individual contributions
while theta < 2*pi:
    ....
print obslocation, Enet ## when loop is finished, print location and Enet
arrow(.....)
```

To doubly indent, select these statements, and on the Format menu choose "Indent Region" twice.

Next, modify your program to place your (now doubly indented) field calculations inside two new loops, so as to calculate and display the electric field at many observation locations in a grid in the xy plane. Your program should look like this:

```
from  __future__  import division
from visual import *
```

```
## constants
... [your constants here, including scalefactor]
## objects
... [your ring here]
## initial values
[your initial values from above]
nx = -12
while nx <= 12: ## run from -12*dx to +12*dx; "<=" means less than or equal
   ny = -3
   while ny <= 3: ## run from -3*dy to +3*dy
         obslocation = vector(nx*dx,ny*dy,0)

         ## calculations
         theta = 0
         Enet = vector(0,0,0) # for adding up the individual contributions
         while theta < 2*pi:
               ....
         arrow(...)

         ny = ny+1
   nx = nx+1
```

Your original calculations are now inside two additional "nested" while loops. The outermost loop, `while nx <= 12:`, runs from -12 to +12, and the next loop, `while ny <= 3:`, runs from -3 to +3. Note the calculation of `obslocation` in terms of nx and ny, and the incrementing of nx and ny by 1 at the end of the nx and ny while loops.

The effect is that the sequence of calculated `obslocation` values looks like this:

< -12*dx, -3*dy, 0 > (with nx = -12, ny runs from -3 to +3)
< -12*dx, -2*dy, 0 >
< -12*dx, -1*dy, 0 >
< -12*dx, 0*dy, 0 >
< -12*dx, +1*dy, 0 >
< -12*dx, +2*dy, 0 >
< -12*dx, +3*dy, 0 >

< -11*dx, -3*dy, 0 > (now with nx = -11, ny again runs from -3 to +3)
< -11*dx, -2*dy, 0 >
etc.

The locations have been chosen to give you a sense of the pattern of electric field near a charged ring

- Run your program. It should calculate and display (as arrows) the electric field at the specified locations near the ring. (You might comment out your print statement to make the program run faster.)
- Adjust `scalefactor` to make sure that no arrow overlaps another arrow.
- Rotate the "camera" with the mouse to get a feeling for the nature of the pattern of electric field.
- Look at your display. Does it make sense? Do the magnitudes and directions of the orange arrows make sense?
- The electric field is zero at the center of the ring (why?). The electric field also approaches zero very far from the ring. Therefore we should expect that along the axis there should be a place where the magnitude of the field is a maximum. Do you see such a maximum on the axis in your display? Another way of seeing this effect is the graph of E vs. z in the textbook on p. 529. This is unusual. Most charge distributions make a field that decreases with distance from the object.
- Change the sign of the charge to negative and run. Does the display make sense?
- Reset the charge to positive.

Make sure everyone in the group agrees that the display looks right.
Check your display with a neighboring group.

5 Number of slices and accuracy

With 20 slices, the numerical integration is probably pretty accurate, but you should check this. Try N = 2, 4, 20, 50, and 100. If you find that increasing N doesn't make a significant change, that value of N is good enough.

What do you find is a "good enough" value of N, approximately, as judged by the pattern of field? Isn't this a bit surprising? Ask a neighboring group what they found to be an adequate number of slices.

- Reset the number of slices to 20 and run your program to make sure it's running properly.
- Temporarily uncomment your print statement and run the program in order to be able to report to WebAssign the electric field you find at a particular specified location, using 20 slices.

Make sure everyone in the group agrees that the display looks right.
Make sure that no arrow overlaps another arrow.
Show your display to your instructor.
Then the Recorder turns in the group's program to WebAssign.

You may be asked to make some changes to the program, and answer some questions about it.

6 Playing around

Here are some suggestions of things you might like to try after turning in your program:

- In the Visual reference manual available on the Help menu, in the section on "Mouse Interactions", find out how to determine the current position of the mouse. Display the electric field at the mouse location, either where you click the mouse, or continuously as you drag the mouse.

- Here's a rather ambitious project, but one you might find interesting. In Python you can declare a subprogram and use that subprogram wherever you want, much as you use sqrt(x) to calculate the square root of x. Take your block of statements that calculate the electric field (no arrow) and put them inside a Python "def" structure ("define" a subprogram) just after your "constants" section:

```
def field(obslocation):
   ## calculations
   theta = 0
   Enet = vector(0,0,0) # for adding up the individual contributions
   while theta < 2*pi:
      ....
   return Enet
```

Anywhere in your program that you want the field at a particular location r, you just say field(r). Python then gives r to the field subprogram, which returns the field value you want. It's the same as the sqrt function; you just say sqrt(x) and Python gives x to the sqrt subprogram, which returns the square root you want. Now you can use field(loc) anywhere in your program, replacing the field calculation statements inside the x and y loops, for example.

But here's where it gets to be interesting. Use the Momentum Principle as you did in mechanics to move an electron near the ring. Make a sphere to represent the electron. To calculate the force on the electron, simply say F = qe*field(electron.pos), where qe is -1.6e-19 C (mass is 9e-31 kg). The field subprogram calculates the field at the current location of the electron, no matter where the electron is. To leave a trail, create trail = curve(), then say trail.append(pos=electron.pos) in the loop.

Here are parameters that give a pretty trajectory. You can probably find even nicer ones.

- initial position = < -12*dx, 0.8*R, 0 >
- initial velocity = < 1e7, 0, 1e7 > (this is only 4% of the speed of light)
- deltat = 1e-11

If you find an interesting trajectory, email your program to your TA. We'd love to see it.