Visualizing Magnetic Fields with VPython

We have already used VPython to map the electric field of a dipole and examined how that field determines the motion of a test charge. Additionally, we have examined the physics behind how a ring of current generates a magnetic field using the Biot-Savart law. Let’s use this knowledge to map the shape of the magnetic field generated by the current-carrying loop in VPython.

Open the item “VIDLE for VPython” in the Start menu to begin the VIDLE editor. We would like our program to have four sections:
- A section defining objects and constants
- A section establishing the locations where we want to map the field at
- A section that actually calculates and displays the magnetic field at those locations
- A section that models the dynamics of a particle in the magnetic field

Variable definitions
Just like you would for any VPython program, type

```
from visual import *
```

to load all the objects pre-defined in the module `visual` that comes with the VPython package. Add an empty line and write a comment that titles your program and lists you as the author. Remember that single line comments must start with a ‘#’ while multiline comments can be created by enclosing any segment of your program with ‘’ (triple quotes, or three apostrophes). Multiline commenting is also useful if you want the computer to ignore a chunk of your program when running it.

Define a set of variables as follows.

```
# Variable definitions
mu0 = 4*pi*1e-7  # Magnetic permeability
Bscale = 1.2e5  # Scale factor for arrows
R = 0.01  # Radius of current loop
I = 1.0  # Current in loop
```

These are mostly self-explanatory parameters needed to define a loop of current and the calculation of its magnetic field. The parameter ‘Bscale’ exists because everything is dimensionless in VPython, so the magnetic field calculated in SI units would have a numerical value orders of magnitude too small (about $10^{-7}$) to represent with arrows on the same image that shows the current loop of radius 0.01.
Now let’s define an object that will be our current loop. Insert blank line below what you’ve written and add this code.

\[ \text{loop} = \text{ring(pos = vector(0,0,0), axis = vector(1,0,0), radius = R, thickness = 0.001)} \]

VPython will draw a ‘ring’-type object of radius ‘R’ centered at the origin with a symmetry axis in the x-direction. Run your program with F5 to see this. You can add the attribute ‘color = color.yellow’ (or replace ‘yellow’ with another color of your choice) to change the color of the ring.

Add a blank line and add yet a few more lines of code to prepare the set of points we want to map.

locations = [ ]  # List of locations to calculate the field at
arrows = [ ]  # List of arrows to plot
dtheta = pi/6.  # Angle intervals between points
dphi = pi/6.
Ro = 6*R  # Radius of the sphere of arrows

Here ‘locations’ will be a list of the points (in vector form), at which we’ll calculate the magnetic field, ‘arrows’ will be a list of the arrows to plot at those points, and ‘dtheta’ and ‘dphi’ will be intervals stepping through points on a sphere that has radius ‘Ro’. Save your program as “CurrentLoop.py”.

**Defining the locations to map the field at**

Let’s create a “shell” of locations at a fixed distance from the ring from the above parameters. Add a new section to your code that begins a loop.

\[ \text{# Define the set of locations for mapping the magnetic field}
\]

\[ \text{for theta in arange(0, pi+dtheta, dtheta):}
\]

\[ \text{\quad x = Ro*cos(theta)} \]
\[ \text{\quad y = Ro*sin(theta)} \]
\[ \text{\quad z = 0} \]
\[ \text{\quad a = vector(x, y, z)} \]
\[ \text{\quad locations.append(vector(a.x, a.y, a.z))} \]

Here we are simply defining coordinates on a circle of radius ‘Ro’ in the x-y plane, assigning those coordinates to a vector ‘a’, and appending that vector to our list of locations. To visualize what we are doing, add the following code after the code above and make sure it is outside of the loop.
# Define a set of default arrows for the field
for point in locations:
    arrows.append(arrow(pos = point, axis = vector(0.01, 0, 0), color = color.magenta, shaftwidth = 0.002))

This code appends an arrow to the list `arrows` for every point in the list `locations`. You should see a semicircle of arrows, all with the same length and direction. If you want to make your default arrows invisible for any reason, you can set their axis to `vector(0, 0, 0)` so that they have zero length.

The function `arange()` in the first loop defines a list of equally-spaced values. Its arguments take the form `(min value, max value, interval)`. Note that the list it generates includes values less than, but not equal to, the max value. Here we want the coordinates to step around the circle from 0 to π in an interval of `dtheta` that we specified above, marking half of the circle, so the max value must be one step larger than π.

Now we can cover the whole sphere by rotating the vector `a` through an angle 2π in intervals of `dphi` in the y-z plane as we step through each `dtheta`. Remove the line `locations.append(vector(a.x, a.y, a.z))` and insert the following code (within the loop!) in its place.

```python
for phi in arange(0, 2*pi, dphi):
    b = rotate(a, angle = phi, axis = vector(1,0,0))
    if b not in locations:
        locations.append(vector(b.x, b.y, b.z))
```

This code rotates the vector `a` by an angle `dphi` around the x-axis, assigns the new vector to the variable `b`, and then assigns the coordinates of `b` to `locations`. The `if/not` statement is to prevent VPython from duplicating the same location more than once in `locations`. Without this, when theta is 0 or π you’d be adding several copies of the same point to the beginning and end of `locations`, since the points on the x-axis do not change when you change phi. Save the latest changes to your program.

**Calculating the magnetic field**

The length and direction of the arrows we inserted above was chosen arbitrarily. What we really want is to make these arrows point in the direction of the magnetic field at that point. To do this we need to write code that effectively integrates the Biot-Savart law around the current loop to find the total magnetic field at a point, and then have that loop over all points. Set the length of the
arrows you defined to ‘axis=((0, 0, 0))’ so that they all disappear. Then initialize a few new variables beneath all of the previous code.

```
# Calculating the magnetic field

dalpha = pi/10

dlarrow = arrow(pos = vector(0,0,0), axis = vector(0,0,0), color = color.red, shaftwidth = 0.005)

rarrow = arrow(pos = vector(0,0,0), axis = vector(0,0,0), color = color.green, shaftwidth = 0.005)
```

Here ‘dalpha’ will be the interval with which we integrate around the loop through the angle ‘alpha’, ‘dlarrow’ defines an arrow along the direction of the loop for the infinitesimal path length ‘dl’, and ‘rarrow’ defines an arrow from the point we’re at on the loop to the point in ‘locations’ where we are calculating the magnetic field. This is the vector ‘r’ in the Biot-Savart law.

```
for Barrow in arrows:
    B = vector(0,0,0)
    for alpha in arange(0, 2*pi, dalpha):
        c = vector(0, R*cos(alpha), R*sin(alpha))
        d = vector(0, R*cos(alpha+dalpha), R*sin(alpha+dalpha))
        dl = d - c
        dlarrow.pos = c
        dlarrow.axis = dl
        r = Barrow.pos - c
        rarrow.pos = dlarrow.pos
        rarrow.axis = r

        B = B + cross(c, r) / (norm(r)**2)

B = B * Bscale
print mag(B)
```

Let’s break down this section of code. The loop is over all the arrows in ‘arrows’ as we want it to be, which the loop refers to as a ‘Barrow’. We initialize the magnetic field ‘B’ to the zero vector and then define a loop that approximates integration over angle ‘alpha’ of the Biot-Savart law around the ring. Here ‘dl’ is defined as the difference of two vectors ‘c’ and ‘d’ that point to locations on the ring separated by an angle ‘dalpha’, and ‘r’ is the vector from the point ‘c’ on the ring to the location of the current ‘Barrow’ in ‘arrows’ whose properties we are going to calculate and assign. The actual step of calculating and updating ‘B’ has been left out so that you can figure
it out for yourself. Then the ‘Barrow’ is assigned the orientation along the vector ‘B’ and scaled to make it visible. Its magnitude is output to the Python shell. Run the program to see what this segment of code does.

Add a line at the very end of the loop over ‘alpha’ (after your Biot-Savart expression to update ‘B’) as follows:

```python
    scene.mouse.getclick()
```

This command will let you click through the loop that you just wrote step by step. You can watch the integration around the ring and then the appearance of each ‘Barrow’ as the loop moves on to the next location. You can switch between clicking through the steps and having the program run all at once by commenting out this line of code. (Remember this! If your program appears to be frozen it may just be that you left this line uncommented when running it!) Save the new additions to your program.

**Motion of a charge in the magnetic field**
Currently we only have a single sphere of arrows to map the field. To add multiple additional spheres of arrows, make some minor changes to the sections where you defined and added elements to ‘locations’. Replace the line ‘R = 6*Ro’ with the following:

```python
    dRo = 0.02
```

This will be the radius increment for our surfaces of arrows. To define the range of Ro, take the ‘for theta’ loop and enclose it entirely in another loop that reads

```python
    for Ro in arange(6*R, 18*R+dRo, dRo):
```

This will now add rings of arrows to ‘locations’ in increments of ‘dRo’ from 6*R out to 18*R. Now we have a series of spherical surfaces on with we have calculated and displayed the magnetic field of the loop. To greatly speed up the time it times your program to run, comment out any lines that print text to the shell.

**Motion of a charge in the magnetic field**
Finally, let’s add the code to have a particle move around in the magnetic field. Beneath the first section of your code define a ‘proton’ object.

```python
    proton = sphere(pos = vector(0.05, 0.03, 0.01), radius = 0.003, color = vector(0, 1, 1), 
    charge = 1.6e-19, m = 1.7e-27, p = vector(-5e-28, -5e-28, 5e-29), trail = curve(color = 
    vector(1, 1, 1)))
```
The position and momentum have been chosen so that they provide an interesting trajectory by default. Now write up the code below to model the dynamics.

```python
# Particle dynamics
dt = .001

while t < 0.3:
    t = t + dt
    B = vector(0,0,0)

    for alpha in arange(0, 2*pi, dalpha):
        c = vector(0,R*cos(alpha), R*sin(alpha))
        d = vector(0,R*cos(alpha+dalpha), R*sin(alpha+dalpha))
        dl = d - c
        dlarrow.pos = c
        dlarrow.axis = dl
        r = proton.pos - c
        rarrow.pos = dlarrow.pos
        rarrow.axis = r

    # Insert the exact same code here that you had to write for the Biot-Savart law in the previous section.

    # Include a line of code that defines the Lorentz force ‘F’ on the particle here
    proton.p = proton.p + F*dt
    proton.pos = proton.pos + (proton.p/proton.m)*dt
    proton.trail.append(proton.pos)
```

After making the appropriate additions to the code above, you will now have a working simulation for a particle in the field of a ring of current. The time interval was again chosen to provide an interesting example by default.

Now you can be creative with the program. Adjust all of the variables to see what effect they have on the simulation. Play around and have fun. Add or do something unique to personalize it before turning it in to your TA for a grade.