Gauss's Law Python

Here we want to calculate the electric flux through a spherical surface for any charge distribution.

Simple Gauss's Law

Suppose I have a point charge (q) at the origin. If we assume the magnitude of the electric field only depends on the distance from the charge and always points radially outward, it might look something like this.

Gauss's Law states:

$$\oint \vec{E} \cdot \hat{n} dA = \frac{q_{\text{in}}}{\epsilon_o}$$

So, if we pick a sphere of radius r centered on the charge the flux becomes trivial since we can write:

$$\vec{E} = E(r)\hat{r}$$

And

$$\oint E(r)\hat{r} \cdot \hat{r} dA = E \oint dA = E4\pi r^2 = \frac{q}{\epsilon_0}$$

This gives the well known expression for the electric field due to a point charge.

$$E = \frac{1}{4\pi\epsilon_0}\frac{q}{r^2}$$

But what if the charge is not in the center of the Gaussian sphere? Or what if the charge is outside of the sphere? Gauss's Law should still work but the integral will no longer be trivial. This means we are going to have to do it numerically.

Building a Sphere in Python

Let's make a sphere of radius R = 0.1 by breaking it into pieces. We are going to describe this sphere with both spherical and Cartesian coordinates (Cartesian coordinates are needed for drawing 3D objects in python). Just a quick reminder that in spherical coordinates, we use the following:

r as the distance from the origin to some point.
θ is the angle from the line to the point and the z-axis.
ϕ is the angle from a projection into the x-y plane and then measured angle to the x-axis.

With that, we get the following relationship.

$$x = r\sin\theta\cos\phi \qquad y = r\sin\theta\sin\phi \qquad z = r\cos\theta$$

Here θ goes from 0 to π and ϕ goes from 0 to 2π.

Now let's say we want to make a sphere by breaking it into little pieces. I can use the angles θ and ϕ to spread these pieces around. Let's do this using tiny spheres as pieces on our surface (just for now, we will fix it later).

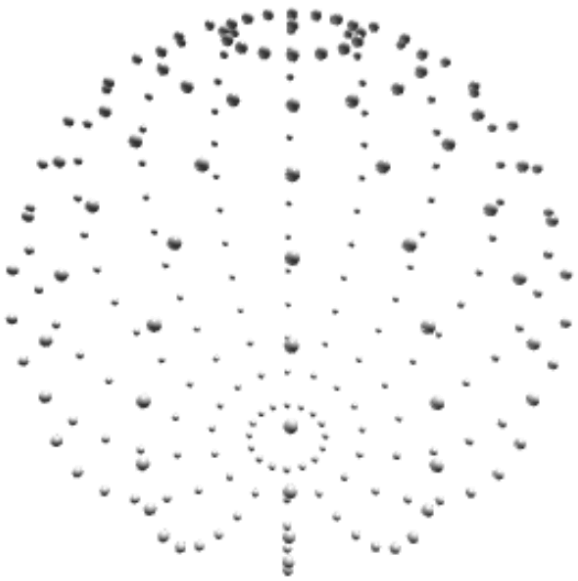
We need to make a double loop over both ϕ and θ. Here is the code.

```
1  Web VPython 3.2
2  canvas(background=color.white)
3  R = 0.1
4  N = 16
5
6  theta = 0
7  dtheta = pi/N
8  phi = 0
9  dphi = 2*pi/N
10
11 while theta<pi:
12   phi = 0
13   while phi<2*pi:
14     ro = vector(R*sin(theta)*cos(phi),R*sin(theta)*sin(phi),R*cos(theta))
15     sphere(pos=ro, radius=R/60)
16     phi = phi + dphi
17   theta = theta + dtheta
18
```

Comments:

- Of course I'm using Web VPython (https://www.glowscript.org). If you want to use real python, don't use line 1 and you will need to both install and then import the vpython module.
- Here, I'm breaking this into 16 pieces in both angular directions.
- In the double loop, make sure to reset the angle value back to zero for the inner loop.

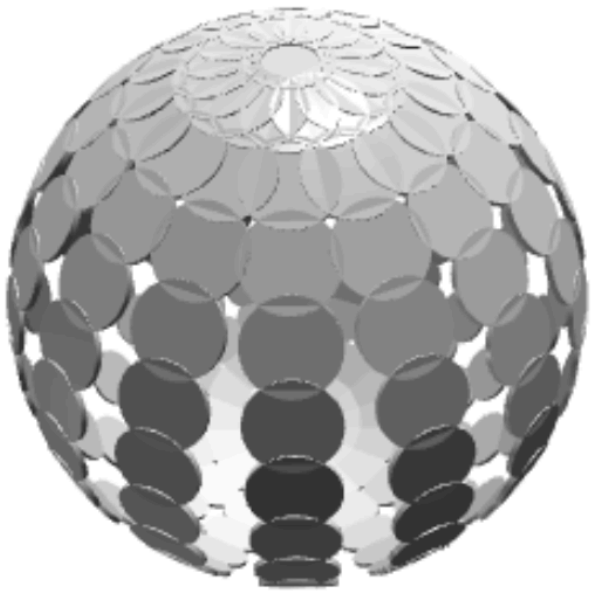
Here is what this looks like (I rotated the 3D object for a better effect)



That's cool, but we need tiny little areas so the we can calculate the electric flux. Let's replace the tiny spheres with tiny circles (they will actually be cylinders). Remember that in VPython, the cylinder object has the following important properties.

- pos. This is the vector position of one end of the cylinder.
- axis. This is a vector from pos to the other end of the cylinder.
- radius. Do I have to explain that the radius is the radius?

The cool thing about the cylinder is that we can have each tiny cylinder such that it's mostly flat on the surface of the sphere. Here's what that would look like (I'll fix the problem then show code later).



Although this looks cool, we clearly can't have all the circles the same size. For values where θ is closer to zero, they should be smaller.

Let's just back to spherical coordinates. If we break things into tiny pieces we get the following volume element.

$$dV = dL_r dL_\theta dL_\phi$$

$$dL_r = dr \qquad dL_\theta = r d\theta \qquad dL_\phi = r \sin\theta d\phi$$

So, with r being constant on the surface, we have the surface area element as:

$$dA = dL_\theta dL_\phi = r^2 \sin\theta d\phi d\theta$$

Using constant values for $d\theta$ and $\mathrm{s}\phi$ means that dA will NOT be constant. We can calculate the area for each piece and then use that to make a circle of the same size.

Here is the new code (well, just part of the code).

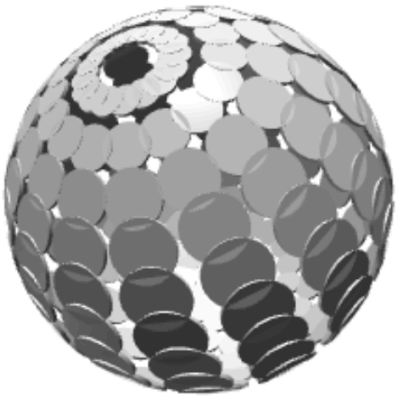
```
while theta<pi:
  phi = 0
  while phi<2*pi:
    ro = vector(R*sin(theta)*cos(phi),R*sin(theta)*sin(phi),R*cos(theta))
    dLtheta = R*dtheta
    dLphi = R*sin(theta)*dphi
    rcirc = sqrt(dLtheta*dLphi/pi)
    cylinder(pos=ro,axis=(R/60)*norm(ro),radius=rcirc)
    phi = phi + dphi
  theta = theta + dtheta

```

Comments:

- In order to get the circle with the correct orientation, the axis is set to norm(ro). The norm() function takes a vector and returns a unit vector in that direction. Since ro is the vector from the center to the disk, norm(ro) as the axis will make the circle flat on the sphere.
- Notice also that the radius of the cylinder is calculated from the area element.

Here's what it looks like now.



Notice that the circles near the "poles" are smaller because the area elements are smaller. Yes, they still overlap some - but that's OK. Ideally, these areas should be more like a square but that makes it more difficult to draw.

Lists and Area Check

Yes, before we get to Gauss's Law let's check the area of our numerical sphere and see if what we expect ($A = 4\pi R^2$). The first thing we are going to do is to add all of our circles to a list. That way we can reuse their values when we need to.

Here is a modification to the code.

```
11  areas = []
12  while theta<pi:
13    phi = 0
14    while phi<2*pi:
15      ro = vector(R*sin(theta)*cos(phi),R*sin(theta)*sin(phi),R*cos(theta))
16      dLtheta = R*dtheta
17      dLphi = R*sin(theta)*dphi
18      rcirc = sqrt(dLtheta*dLphi/pi)
19      areas = areas + [cylinder(pos=ro,axis=(R/60)*norm(ro),radius=rcirc,
20      dA = dLtheta*dLphi)]
21      phi = phi + dphi
22    theta = theta + dtheta
23
24  A = 0
25  for a in areas:
26    A = A + a.dA
27  print("surface area = ",A)
28  print(4*pi*R**2)
29
```

Comments:

- Line 11: here I create an empty list called “areas”.
- Line 19: instead of just drawing the cylinders, I’m adding them to the areas list. Note this line wraps around to line 20.
- Line 19-20: I added a property to each cylinder. That property is called dA (the area). This is useful later.
- Lines 24-26: This calculates the total area. We start off with a value of A = 0 so that we can add to this. Line 25 goes through the elements of the areas list can calls each element “a”. This means that a.dA is the area of that circle so I can add it to the total.
- At the end, I print the calculated area and the theoretical area. Both are right around 0.125 so it works.

Electric Flux for Center Charge

Now for the physics. Suppose I place a charge a vector location $\vec{r}_{q1}$ that has a charge q. If we want to calculate the electric field at an observation location $\vec{r}_o$ we would do the following.

$$\vec{r} = \vec{r}_o - \vec{r}_{q1}$$

$$\vec{E} = \frac{1}{4\pi\epsilon_o}\frac{q}{|\vec{r}|^2}\hat{r}$$

Once we have the electric field at the location of each tiny area, we can assume the value of the field is constant over that area. With that, the tiny flux over that tiny area would be:

$$d\Phi = \vec{E} \cdot \hat{n} dA$$

Here $\hat{n}$ is a vector perpendicular to the area (which I already know from drawing the circles as little cylinders. Then Gauss's Law would look like this.

$$\Phi = \sum_i d\Phi = \frac{q_{\text{in}}}{\epsilon_o}$$

So, here's what needs to be done:

- Go through the list of areas.
- For each area (circle on the sphere) calculate the electric field at that location.
- Using the electric field and the axis for that cylinder, calculate the flux.
- Add up all the fluxes.

Simple, right? Here's the code.

```
11  areas = []
12  eps = 1 #this is epsilon0
13  k = 1/(4*pi*eps)
14  q1 = sphere(pos=vector(0,0,0),radius=R/50, color=color.red,q=1)
15
16  while theta<pi:
17    phi = 0
18    while phi<2*pi:
19      ro = vector(R*sin(theta)*cos(phi),R*sin(theta)*sin(phi),R*cos(theta))
20      dLtheta = R*dtheta
21      dLphi = R*sin(theta)*dphi
22      rcirc = sqrt(dLtheta*dLphi/pi)
23      areas = areas + [cylinder(pos=ro,axis=(R/60)*norm(ro),radius=rcirc,
24      dA = dLtheta*dLphi)]
25      phi = phi + dphi
26    theta = theta + dtheta
27
28  Flux = 0
29  for a in areas:
30    r = a.pos - q1.pos
31    E = k*q1.q*norm(r)/mag(r)**2
32    dFlux = dot(E,norm(a.axis))*a.dA
33    Flux = Flux + dFlux
34  print("Flux  = ",Flux)
35  print("q in/eps = ",q1.q/eps)
36
```

Comments:

- I made the charge as a sphere (called it q1) and gave it a charge property q = 1.
- Oh, for the units, I figured it's cleaner to use $\epsilon_o = 1$.
- Line 28: to calculate the flux, I need to start with flux = 0 so that I can add to it for each circle.

- Lines 30-31: This calculates the electric field at the circle. Notice that r is the vector from the charge to the area.
- Line 32: Calculate the flux. Here we are using the built in dot product function. I have norm(a.axis) because I don't want to use the actual length of the cylinder. Instead, I multiply this by the circle's area.

Go ahead and run this and see if Gauss's Law works.

Let's Make It Pretty

I am going to do the following:

- Make the circles partially transparent.
- Change the color of the circles - red for positive flux and blue for negative flux.
- Add arrows for the electric field.

Here's the addition to the code for the circles.

```
22      rcirc = sqrt(dLtheta*dLphi/pi)
23      areas = areas + [cylinder(pos=ro,axis=(R/60)*norm(ro),radius=rcirc,
24      dA = dLtheta*dLphi,opacity=0.4, color=color.white)]
```

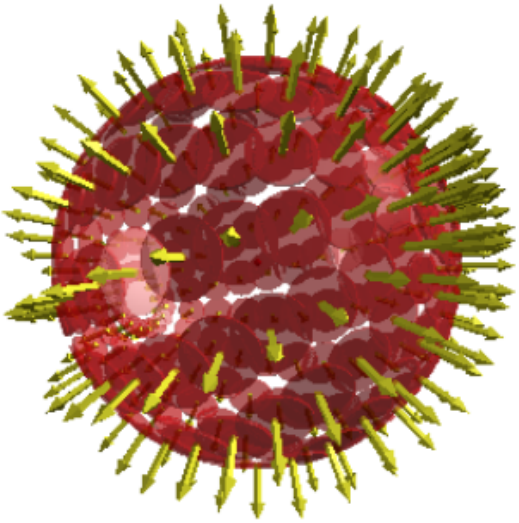
The opacity is the level of transparency. I'm starting off with the color as white so that I can change it later.

For the electric field arrows, we need a scaling factor (calling it Escale). Also, I just need to do an if statement to see what color to make each circle.

```
30 for a in areas:
31   r = a.pos - q1.pos
32   E = k*q1.q*norm(r)/mag(r)**2
33   arrow(pos=a.pos, axis=Escale*E,color=color.yellow)
34   dFlux = dot(E,norm(a.axis))*a.dA
35   if dFlux>0:
36     a.color=color.red
37   if dFlux<0:
38     a.color=color.blue
39   Flux = Flux + dFlux
```

I'm using an Escale value of 0.003 - but you pick a value that make the field look nice for you.

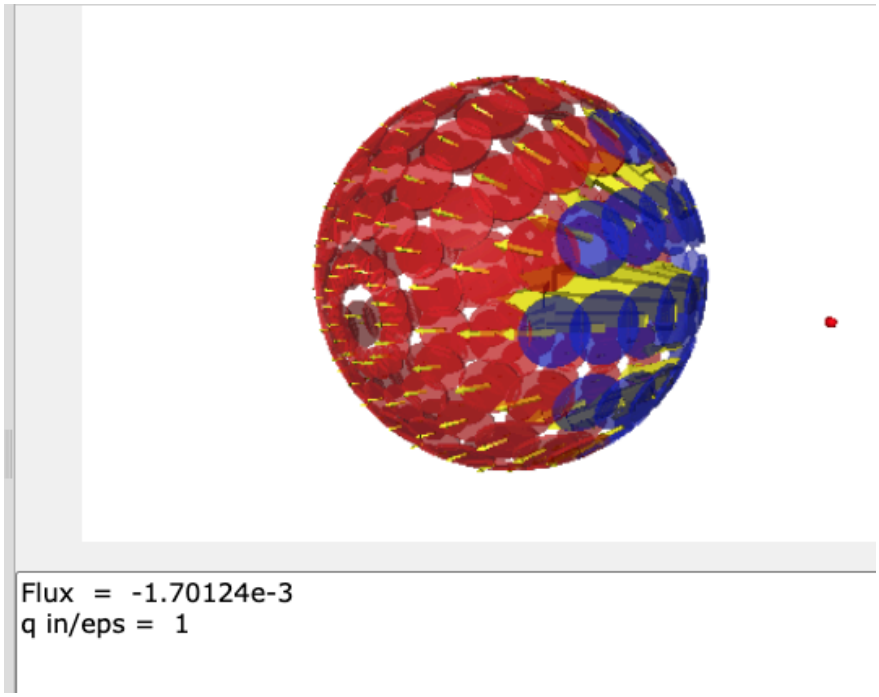
Here's the output.



That's cool.

Non Symmetrical Situations

What happens if we move the charge outside the Gaussian sphere? That's pretty simple to see. We just need to change the vector position of q1. Here's what it looks like.



Notice that the flux is now very close to zero - as it should be.