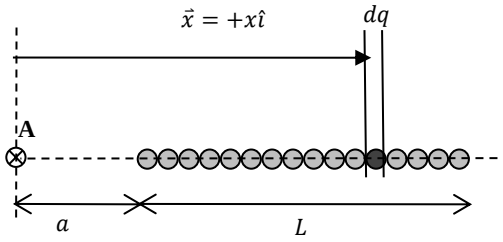
Coding continuous distributions of charge

We will start with a rod shape as shown in the figure at right. From the workbook problem **22.10** the electric field at the origin is given by

$$\vec{E}_{@origin} = -\frac{kQ}{a(L+a)}\hat{\imath}$$



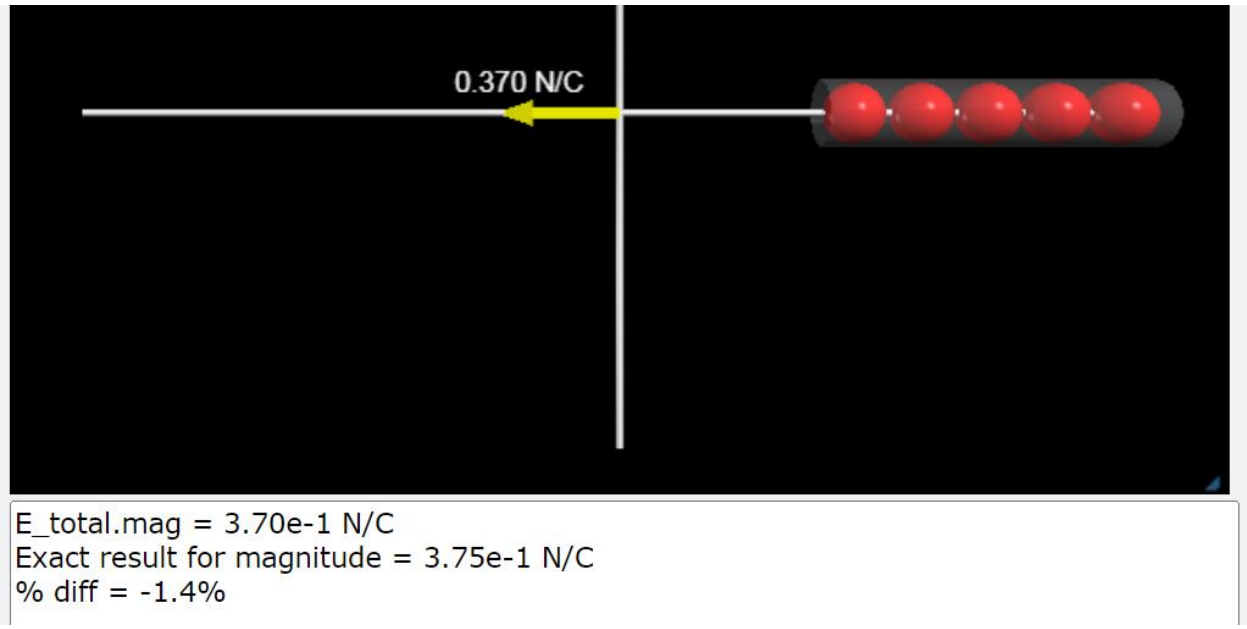
The following page sketches out how you might calculate the electric field due to a continuous distribution of charge. We want to use the answer shown above to check our code. This means we want to define quantities in the code in terms of the given constants.

For instance:

- The charge on a single piece of the rod is given by $dq = \lambda\, dx$.
- If we use N balls to build the rod, we should use $dx = \frac{L}{N}$.
- Charge density will be given by $\lambda = \frac{Q}{L}$.

If one types in constants for $N, L, \& Q$ we should be able to have the computer determine the charge on each ball dq!

On the next few pages I will show screen shots of code I wrote to produce the image shown below.
Your first goal is to follow along, type in the code, and verify it produces the same results.
Then you will modify the code to handle non-uniform cases for both rods and arcs.



First let's get the constants and some cosmetic stuff typed in.

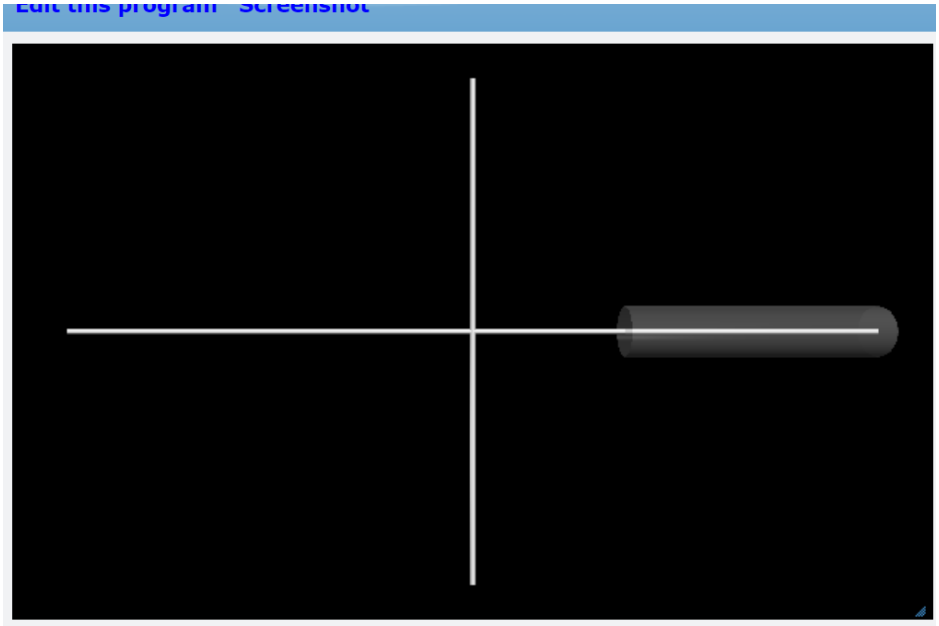
Pay attention to line numbers so your work follows my work closely.

```
1  Web VPython 3.2
2
3  k = 8.99e9
4  POI = vec(0,0,0)          #defines point of interest for E-field computation
5  Q = 1e-9                  #charge on rod in C
6  a = 3                     #offset of left end of rod in meters
7  L = 5                     #rod length in meters
8  N = 5                     #number of spheres used to draw the rod
9  dx = L/N                  #centers of each sphere should be dx apart
10 lambda = Q/L              #charge density on rod
11 scale_factor = 5          #scales size of drawn E-field arrows
12
13 rod = []                  #empty list to store created spheres
14
15 #this draws a semi-transparent cylinder to repesent the charged rod
16 visual_guide = cylinder( pos=vec(a,0,0),
17                           axis=vec(L,0,0),
18                           radius=0.5,
19                           opacity=0.25 )
20
21 #draw a set of coordinate axes
22 line_x=cylinder( pos=vec(-8, 0, 0), axis=vec(16, 0, 0), radius=0.05 )
23 line_y=cylinder( pos=vec(0, -5, 0), axis=vec(0, 10, 0), radius=0.05 )
24
```

Be sure to run the code before proceeding.

This allows you to check for typos more rapidly.

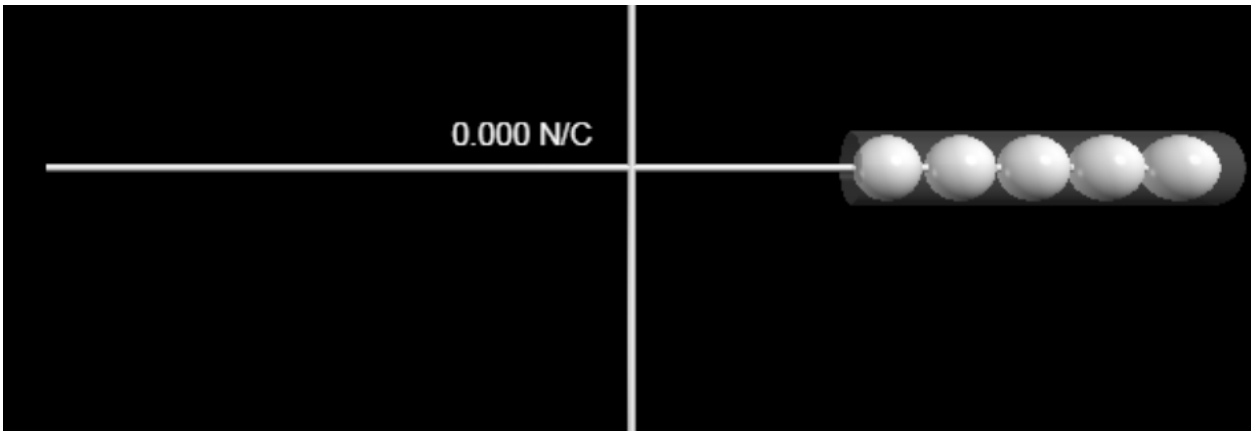
You should see something like this:



Now add this bit of code. Again, watch line numbers closely to simplify communication.

```
24
25 #Following 3 lines of code initalize parameters for integration in FOR loop.
26 x_min = a
27 x_max = L + a
28 E_total = vec(0,0,0)
29
30 E_arrow = arrow( pos=POI, axis=scale_factor*E_total, color=vec(1,1,0) )
31 E_label = label( pos=POI, xoffset=-14, yoffset=14, box=False, opacity=0 )
32 E_label.line = False
33 E_label.text = f"{E_total.mag:.3f} N/C"
34
35 for x in arange(x_min, x_max, dx):
36     rate(4)
37
38     #this shift ensures all drawn spheres spread equally through out rod
39     #without this shift, sphers are skewed left and % diff goes way up
40     shifted_x = x + 0.5*dx
41
42     ball = sphere( pos=vec(shifted_x,0,0),
43                    radius = 0.45,
44                    dq = lambda*dx)
45
```

Run the code to verify you see this output.



Next code fragment actually computes the electric field.
Lines 46 thru 49 will color the charges differently based on the sign of Q.

```
45
46      if ball.dq >= 0 :
47          ball.color=vec(1,0,0)
48      else:
49          ball.color=vec(0, 0.4, 1)
50
51      #this adds each sphere (with all attributes) to the initially empty list
52      #this step doesn't seem important now, but it turns out to be useful later
53      rod.append(ball)
54
55      #following lines compute r-vector from each dq to POI
56      #then adds an E-field contribution to E_total while updating arrow & text
57      r = POI - ball.pos
58      E_total += k*ball.dq*r / r.mag**3
59      E_arrow.axis = scale_factor*E_total
60      E_label.text = f"{E_total.mag:.3f} N/C"
61
```

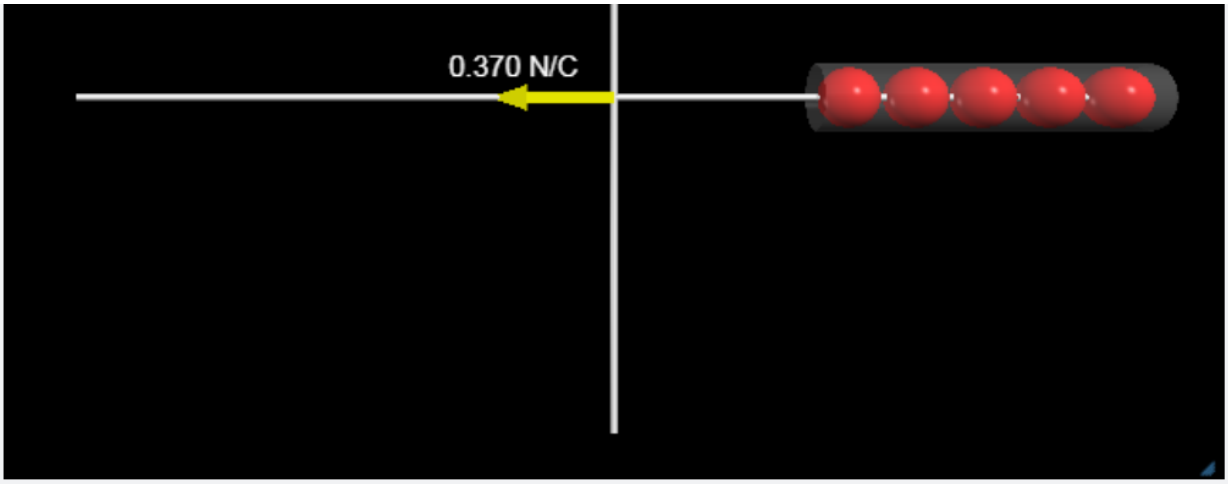
If $Q > 0$ you should see the following output.
For fun, see what happens if you use $Q < 0$.

So far so good….but we can do a lot better with minimal effort.

The next bit of code will show you how to compare the code's result for $\vec{E}_{total}$ to the exact result from calculus.

```
61
62 #notice i am no longer indented...this part of code will run AFTER the FOR loop
63 print(f"E_total.mag = {E_total.mag:.2e} N/C")
64
65 E_mag_exact = k*Q / ( a*(L+a) )
66 def percent_difference(exp, th):
67     p_diff = 100*(exp-th) / th
68     return p_diff
69
70 print(f"Exact result for magnitude = {E_mag_exact:.2e} N/C")
71 print(f"% diff = {percent_difference(E_total.mag, E_mag_exact):.1f}%")
72
```

Notice the code output now gives important text output below the visuals.




```
E_total.mag = 3.70e-1 N/C
Exact result for magnitude = 3.75e-1 N/C
% diff = -1.4%
```

Something important I hope you take away from this lab:

> Never trust your codes until you check them against simple problems and verifying results with a percent difference.
>
> Furthermore, the percent difference should decrease as we ask the code to perform more calculations (as $N \rightarrow \infty$).

Students have the tendency to believe their codes are working if the arrows point in the correct direction. This is a crucial *qualitative* check to be sure. However, in a real life situation you want as many ways to check your work as possible. That means checking your codes *quantitatively* against as many known test cases as possible before letting it rip and using the code to produce new results that are impossible to test.

How to handle non-uniform distributions

Consider the following calculus problems shown in the table below.

	Using Calculus	Using Code
Uniform distribution	$\lambda = \frac{Q}{L}$ is constant $\vec{E} = \int_i^f \frac{k\ dq\ \vec{r}}{r^3}$ $\vec{E} = \int_i^f \frac{k\ \lambda\ dx\ \vec{r}}{r^3}$ $\vec{E} = \lambda \int_i^f \frac{k\ \ dx\ \vec{r}}{r^3}$	Because λ is a constant, we can define *outside* of the FOR loop. This reduces memory usage since we only need to compute it one time. In fact, we could also define dq outside of the loop since each sphere has the same size and thus the same charge! Unfortunately, in our code defining dq outside the loop doesn't give us significant benefits.
Non-uniform distribution	$\lambda = \lambda(x)$ some function of x $\vec{E} = \int_i^f \frac{k\ dq\ \vec{r}}{r^3}$ $\vec{E} = \int_i^f \frac{k\ \lambda\ dx\ \vec{r}}{r^3}$ $\vec{E} = \int_i^f \frac{k\ \lambda(x)\ dx\ \vec{r}}{r^3}$	Because $\lambda(x)$ is NOT constant, we must define *inside* the FOR loop. This reduces memory usage since we only need to compute it one time.

Next page will show how I incorporated this information into the code…

Notice the changes in the code shown below to accommodate the non-uniform density.
Notice I defined new constants & removed comments which shifted line numbers!!! **You need to figure this out!!!**
The solution to workbook problem **22.14** shows how I used $Q_{total} = \int dq$ to relate α to Q & L.

Also, notice the second FOR loop.
Because we appended each sphere into the rod, we can now do all the cosmetic work in a separate loop.
It is probably good form to do computational work in one loop and cosmetic/visualization stuff in another.

Finally, THINK about the percent difference calculation!
Because we have a new problem, you need to modify the code's computation for `E_mag_exact`!
Use the solution in the workbook to update the way the code computes the percent difference.
Yeah, it's long and ugly, but split it into manageable pieces and have the computer do the heavy lifting!

```
37 for x in arange(x_min, x_max, dx):
38     rate(4)
39     #this shift ensures all drawn spheres spread equally through out rod
40     shifted_x = x + 0.5*dx
41 
42     lambda = alpha*shifted_x**2
43 
44     ball = sphere( pos=vec(shifted_x,0,0),
45                    radius = 0.45,
46                    dq = lambda*dx)
47 
48     if ball.dq > dq_max: dq_max=ball.dq
49     if ball.dq < dq_min: dq_min=ball.dq
50 
51     #adds each sphere (with all attributes) to the initially empty list
52     rod.append(ball)
53 
54     #compute r-vec from dq to POI, add E-field contribution, update arrow & text
55     r = POI - ball.pos
56     E_total += k*ball.dq*r / r.mag**3
57     E_arrow.axis = scale_factor*E_total
58     E_label.text = f"{E_total.mag:.3f} N/C"
59 
60 #do all cosmetic work in this loop (ball colors & opacities)
61 for my_ball in rod:
62     rate(4)
63 
64     if my_ball.dq >= 0: my_ball.color=vec(1,0,0)
65     else: my_ball.color=vec(0, 0.4, 1)
66 
67     if my_ball.dq >= 0: my_ball.opacity = my_ball.dq / dq_max
68     else:  my_ball.opacity = my_ball.dq / dq_min
```

How to handle arcs

Consider the following changes to the constants and comments! Notice lines 15-17 are left blank for you to figure out! Notice in line 20 the name of the list has changed!

```
1  Web VPython 3.2
2
3  k = 8.99e9
4  POI = vec(0,0,0)         #defines point of interest for E-field computation
5  Q = 1e-9                 #charge on rod in C
6  R = 5                    #radius of arc
7  theta_min = radians(30) #starting angle for arc converts 30 degrees to radians
8  theta_max = radians(150)#ending angle for arc converts 120 degrees to radians
9
10 #notice the following variable names are so obvious no comment is required
11 theta_total = theta_max - theta_min
12 arc_length_total = R*theta_total
13
14 N = 8                        #number of spheres used to draw the rod
15 dtheta =                     #ANGULAR separation of spheres
16 ds =                         #ARCLENGTH separation of spheres
17 lambda =                     #charge density on arc
18 scale_factor = 5             #scales size of drawn E-field arrows
19
20 arc = []                     #empty list to store created spheres
```

Consider the changes to the loop shown below. Notice you should figure out info for lines 38 & 46.
Also, pay attention to the name of the list in line 54!!!
Finally, don't forget the new theoretical value for comparison in the % difference (see Wkbk sol'n **22.23**).

```
34 for theta in arange(theta_min, theta_max, dtheta):
35     rate(4)
36
37     #this shift ensures all drawn spheres spread equally through out rod
38     shifted_theta =
39
40     #it is good form to avoid overly long lines of code
41     #the next line helps us keep line
42     shifted_pos = R*vec( cos(shifted_theta), sin(shifted_theta), 0)
43
44     ball = sphere( pos=shifted_pos,
45                    radius = 0.45,
46                    dq =          )
47
48     if ball.dq >= 0 :
49         ball.color=vec(1,0,0)
50     else:
51         ball.color=vec(0, 0.4, 1)
52
53     #this adds each sphere (with all attributes) to the initially empty list
54     arc.append(ball)
55
56     #compute r-vec from dq to POI, add E-field contribution, update arrow & text
57     r = POI - ball.pos
58     E_total += k*ball.dq*r / r.mag**3
59     E_arrow.axis = scale_factor*E_total
60     E_label.text = f"{E_total.mag:.3f} N/C"
```

How to handle arcs when using a coordinate shift

Consider the coordinate system in workbook problem 22.14.
Because we have a non-uniform charge distribution based on an angle to the vertical axis, we should think carefully about how we must change the code!

Look at page 25 of the workbook.
Look carefully at CHOICE 2.
Think about how you would define the location of dq (e.g. does the x-coordinate use sine or cosine?).
Also the appropriate limits for `theta_min` & `theta_max` based on this new coordinate system.