Computational Modeling in Introductory Physics

June 2012

Mount San Antonio College

Laboratory Materials adapted by Martin Mason from work by Bruce Sherwood, Ruth Chabay and countless others in the Vpython community. All materials are available on the conference memory stick.

Installing Vpython on your computer:

1. Navigate to <http://vpython.org/>

1. Select your operating system

1. Download and install Python-2.7.1
2. Download and install Vpython-xxx-Py2.7-5.72

1. Double click on the VIDLE for VPython Icon to start the program.

1. Open an example program -- for example, bounce2.py.
2. Press F5 to run (or use the Run menu).

Introduction to 3D Computer Modeling

**OBJECTIVES**

In this course you will construct computer models to:

• Visualize motion in 3D using a programming environment called VPython, which is the widely-used Python programming language (python.org) plus a 3D graphics module, visual

• Visualize vector quantities like position, momentum, and force in 3D

• Do calculations based on fundamental principles to predict the motion of interacting objects

• Animate the predicted motions in 3D

In this Lab you will learn:

• How to use VIDLE, the interactive editor for VPython

• How to structure a simple computer program

• How to create 3D objects such as spheres and arrows

• How to use vectors in VPython

**TIME**

You should finish this part of the lab in 50 minutes.

**OVERVIEW OF A COMPUTER PROGRAM**

• A computer program consists of a sequence of instructions.

• The computer carries out the instructions one by one, in the order in which they appear, and stops when it reaches the end.

• Each instruction must be entered exactly correctly (as if it were an instruction to your calculator).

• If the computer encounters an error in an instruction (such as a typing error), it will stop running and print a red error message.

**1** **Using the VIDLE program editor to create a program**

Open the VIDLE program editor by clicking on the ``VPython'' shortcut located on your desktop.

**1.1** **Starting a program: Setup statements**

 **Enter the following statement in the editor window:**

**from visual import \***

Every VPython program begins with this setup statement. It tells the program to use the 3D module (called ``visual''). The asterisk means, ``Add to Python all of the features available in the visual module''.

Before we write any more, let's save the program:

• **In the editor, from the ``File" menu, select ``Save." Browse to a location where you can save the file and give it the name "vectors.py". YOU MUST TYPE the ``.py" file extension -- the editor will NOT automatically add it. Without the ``.py" file extension the editor won't colorize your program statements in a helpful way.**

 Both Python and VPython are undergoing continuous improvement. For example, before Python version 3.0, 1/2 was truncated to zero, but beginning with Python 3.0, 1/2 means 0.5. If you are using a version of Python earlier than 3.0, you should place the following statement as the first statement in your program, before the import of visual:

**from \_\_future\_\_ import division**

This statement (from *space* **underscore underscore** future **underscore underscore** *space* division) tells the Python language to treat 1/2 as 0.5. You don't need this statement if you are using Python 3.0 or later, but it doesn't hurt, because it is simply ignored by later versions of Python.

**2** **3D Objects**

Watch **VPython Instructional Videos: 1. 3D Objects** (http://www.youtube.com/VPythonVideos) demonstrating how to easily create 3D objects in VPython.

• **Complete the challenge task mentioned at the end of the video. Feel free to use any of the information in the video to complete the challenge.**

 **Checkpoint: ASK THE INSTRUCTOR TO LOOK OVER YOUR WORK.**

**2.1** **The 3D graphics scene**

By default the origin  is at the center of the scene, and the ``camera" (that is, your point of view) is looking directly at the origin.

• **Hold down both buttons and move the mouse up and down to make the camera move closer or farther away from the center of the scene. (On a Macintosh, hold down the Options key and the mouse button while moving the mouse.)**

• **Hold down the right mouse button alone and move the mouse to make the camera ``revolve" around the scene, while always looking at the center. (On a Macintosh, hold down the Apple Command key and the mouse button while moving the mouse.)**

When you first run the program, the coordinate system has the positive x direction to the right, the positive y direction pointing up, and the positive z direction coming out of the screen toward you. You can then rotate the camera view to make these axes point in other directions.

**2.2** **Autoscaling and units**

VPython automatically ``zooms" the camera in or out so that all objects appear in the window. Because of this ``autoscaling", the numbers for the ``pos" and ``radius" could be in any consistent set of units, like meters, centimeters, inches, etc. For example, we could have a sphere with a radius of 0.20 m and a position vector of  m. In this course we will always use SI units in our programs (``Systeme Internationale", the system of units based on meters, kilograms, and seconds).

**2.3** **The Python Shell window is important -- Error messages appear here**

***IMPORTANT: Arrange the windows on your screen so the Shell window is always visible.***

***DO NOT CLOSE THE SHELL WINDOW.***

 **KILL the program by closing only the graphic display window.**

Alternatively, simply rerunning your program will kill the graphics window and create a new one.

**2.4** **Scaling Arrows and Comment lines (lines ignored by the computer)**

Comment lines start with a # (pound sign).

A comment line can be a note to yourself, such as:

# objects created in the following lines

Or a comment can be used to remove a line of code temporarily, without erasing it.

You can also put a comment at the end of a line: sphere() # it's round.

**Comment out all but one arrow in your program. For the remaining arrow:**

• **Change something in the arrow's code such that the arrow is half as long and points in the opposite direction, with its tail remaining on the same sphere.**

**3** **Debugging Syntax Errors**

Watch **VPython Instructional Videos: A. Debugging Syntax Errors**

<http://www.youtube.com/VPythonVideos> which discusses common syntax errors produced by novice users of VPython.

**4** **Variable Assignment**

Watch **VPython Instructional Videos: 2. Variable Assignment**

<http://www.youtube.com/VPythonVideos> which demonstrates how to create variables to store information and reference it later.

• **Complete the challenge task mentioned at the end of the video. Assign each 3D object a variable name. Feel free to use any of the information in the video to complete the challenge.**

• **Move one sphere twice as far from the y-axis. What happened to the three arrows?**

**4.1** **Print Command**

• **Start a new line at the end of your program and type:**

**print(*variable.attribute*)**

Replace *variable.attribute* with the name of one of your 3D objects and one of the valid attributes associated with that object. For example, if you want to print the position attribute of a sphere named ball, it would look like this:

print(ball.pos)

• **Run the program.**

• **Look at the Shell window. The printed value should be the same as the value of the attribute you printed.**

 **Checkpoint: ASK THE INSTRUCTOR TO LOOK OVER YOUR WORK. Make sure you are using variable references when defining the attributes of your arrows.**

Another example of change in Python is that before Python 3.0, you could say print ball.pos, but starting with Python 3.0 one must say print(ball.pos). If you are using an earlier version of Python, it is a good idea to use parentheses anyway, because it doesn't hurt, and it works with later versions of Python.

**5** **Document your work.**

Post screen shots of your code and the resulting display windows to your blog for the program you wrote to generate multiple arrows.

**6** **Using VPython outside of class**

You can download VPython from http://vpython.org and install it on your own computer. VPython is also available in the campus public clusters.

**7** **Reference manual and programming help**

There is an on-line reference manual for VPython. In the text editor (VIDLE), on the Help menu, choose ``Visual" for information about 3D objects, or choose ``Python Docs" to obtain detailed information on the Python programming language upon which VPython is based. We will use only a small subset of Python's extensive capabilities.

Computer Models of Motion: Iterative Calculations

**OBJECTIVES**

In this activity you will learn how to:

• Create 3D box objects

• Update the position of an object iteratively (repeatedly) to animate its motion

• Update the momentum and position of an object iteratively (repeatedly) to predict its motion

**TIME**

You should plan to finish this activity in 65 minutes or less.

**COMPUTER PROGRAM ORGANIZATION**

A computer program consists of a sequence of instructions.

The computer carries out the instructions one by one, in the order in which they appear, and stops when it reaches the end.

Each instruction must be entered exactly correctly (as if it were an instruction to your calculator).

If the computer encounters an error in an instruction (such as a typing error), it will stop running and print a red error message.

A typical program has four sections:

• Setup statements

• Definitions of constants (if needed)

• Creation of objects and specification of initial conditions

• Calculations to predict motion or move objects (done repetitively in a loop)

**1** **Setup statements**

Using VIDLE for VPython, create a new file and save it to your own space. Make sure to add ``.py'' to the file name.

 **Enter the following statement in the editor window:**

**from visual import \***

Every VPython program begins with this setup statement. It tells the program to use the 3D module (called ``visual''). The asterisk means, ``Add to Python all of the features available in the visual module''.

Both Python and VPython are undergoing continuous improvement. For example, before Python version 3.0, 1/2 was truncated to zero, but beginning with Python 3.0, 1/2 means 0.5. If you are using a version of Python earlier than 3.0, you should place the following statement as the first statement in your program, before the import of visual:

**from \_\_future\_\_ import division**

This statement (from *space* **underscore underscore** future **underscore underscore** *space* division) tells the Python language to treat 1/2 as 0.5. You don't need this statement if you are using Python 3.0 or later, but it doesn't hurt, because it is simply ignored by later versions of Python.

**2** **Constants**

Following the setup section of the program you would define physics constants. We'll talk about this in later projects.

**3** **Creating an object**

 **Create a box object to represent a track:**

**track = box(pos=vector(0, -0.025, 0), size=(2.0, 0.05, 0.10), color=color.white)**

**Run the program** by pressing F5 (this may be fn-F5 on a Macintosh, depending on how you have set your preferences).

Arrange your windows so the Python Shell window is always visible.

**Kill the program** by closing the graphic display window.

 **Create a second box object to represent a cart:**

Name this object``cart'', with some color other than white. Give this object a position (pos) of vector(0, 0.2, 0) and a size of (0.1, 0.04, 0.06).

**Run the program** by pressing F5. Zoom (both mouse buttons down; hold down Options key on Macintosh) and rotate (right mouse button down; hold down Apple Command key on Macintosh) to examine the scene. The cart should be floating just above the track. Is it? If you don't see two objects, you skipped something.

**Reposition the cart so its left end is aligned with the left end of the track.**

**To do this you will have to answer the following questions:**

1. **Where is the ``pos'' of a box object? The left end? The right end? The center?**
2. **Do the numbers in the ``size'' of a box refer to the total length, or the distance from the center to one edge?**

**You can answer these by experimentation, or by looking in the online reference manual (Help menu, choose Visual).**

**3.1** **Initial conditions**

Any object that moves needs two vector quantities declared before the loop begins:

 1. initial position; and

 2. initial momentum.

You've already given the cart an initial position at the left end of the track. Now you need to give it an initial momentum. If you push the cart with your hand, the initial momentum is the momentum of the cart just *after* it leaves your hand. At speeds much less than the speed of light the momentum is , and we need to tell the computer the cart's mass and the cart's initial velocity.

• **Below the existing lines of code, type the following new lines:**

**mcart = 0.80**

**pcart = mcart\*vector(0.5, 0, 0)**

**print(`cart momentum =', pcart)**

We have made up a new variable name ``**mcart**.'' The symbol ``**mcart**'' now stands for the value 0.80 (a scalar), which represents the mass of the cart in kilograms.

We have also created a new variable **pcart** to represent the momentum of the cart. We assigned it the initial value of (0.80 kg) m/s.

• **Run the program. Look at the Python Shell window. Is the correct value of the vector pcart printed there? From what is printed, how can you tell it is a vector?**

**3.2** **Time step and total elapsed time**

To make the cart move we will use the position update equation  repeatedly in a ``loop''. We need to define a variable **deltat** to stand for the time step , and a variable **t** to stand for the total time elapsed since the motion started. Here we will use the value  s.

• **Type the following new lines at the end of your program:**

**deltat = 0.01**

**t = 0**

This completes the first part of the program, which tells the computer to:

 1. Create numerical values for constants we might need (none were needed this time)

 2. Create 3D objects

 3. Give them initial positions and momenta

**4** **Beginning Loops**

Watch **VPython Instructional Videos: 3. Beginning Loops**

http://www.youtube.com/VPythonVideos

Complete the following task:

• **Create a loop that prints the variable t from 0.0 to 0.19 in increments of deltat (section 3.2)**

• **If you're stuck, watch the video again.**

• **Add a print command after the loop to print the text `End of the loop.'**

• **Run the program. Look at the Python Shell window.**

**5** **Loops and Animation**

Watch **VPython Instructional Videos: 4. Loops and Animation**

<http://www.youtube.com/VPythonVideos> to see how using loops can animate 3D objects. We will use loops and physics principles to build models of motion that are consistent with the natural world. The next section introduces how to translate physics principles into syntax understood by the computer program.

**5.1** **Cart with constant momentum**

Consider a cart moving with constant momentum. Somebody or something gave the cart some initial momentum. We're not concerned here with how it got that initial momentum. We'll predict how the cart will move in the future, *after* it acquired its initial momentum.

You will use your iterative calculation ``loop''. Each time the program runs through this loop, it will do two things:

 1. Use the cart's current momentum to calculate the cart's new position

 2. Increment the cumulative time **t** by **deltat**

You know that the new position of an object after a time interval  is given by



where  is the final position of the object, and  is its initial position. If the time interval  is very short, so the velocity doesn't change very much, we can use the initial or final velocity to approximate the average velocity.

Since at low speed , or , we can write



We will use this equation to increment the position of the cart in the program. First, we must translate it so VPython can understand it.

• **Delete or comment out the line inside your loop that prints the value of t.**

• **On the indented line after the ``while'' statement, and before the statement updating t, type the following:**

**cart.pos = cart.pos + (pcart/mcart)\*deltat**

Notice how this statement corresponds to the algebraic equation:

 **Think about the situation and answer the following question:**

**What will the elapsed time t be after moving two meters?**

• **Change the while statement so the program runs just long enough for the cart to travel 2 meters.**

• **Now, run the program. What do you see?**

***Slowing down the animation***

When you run the program, you should see the cart at its final point. The program is executed so rapidly that the entire motion occurs faster than we can see, because a ``virtual time'' in the program elapses much faster than real time does. We can slow down the animation rate by adding a ``rate'' statement.

• **Add the following line inside your loop (indented):**

**rate(100)**

Every time the computer executes the loop, when it reads ``**rate(100)**'', it pauses long enough to ensure the loop will take  of a second. Therefore, the computer will only execute the loop 100 times per second.

• **Now run the program.**

You should see the cart travel to the right at a constant velocity, ending up 2 meters from its starting location.

**Note:** The cart going beyond the edge of the track isn't a good simulation of what really happens, but it's what we told the computer to do. There are no ``built-in'' physical behaviors, like gravitational force, in VPython. Right now, all we've done is told the computer program to make the cart move in a straight line. If we wanted the cart to fall off the edge, we would have to enter statements into the program to tell the computer how to do this.

 **Answer the following question:**

**Which statement in your program represents the position update formula?**

C**hange your program so the cart starts at the right end of the track and moves to the left.**

When you have succeeded, compare your program to that of another group.

**5.2** **Changing momentum**

Your running program should now have a model of a cart moving at constant velocity from right to left along a track.

• **What should happen to the motion of the cart if you apply a constant force to the right?**

**Discuss this among your group, and write down your prediction.**

As discussed in Class, an iterative prediction of motion can include the effects of forces that change the momentum of an object:

 • Calculate the (vector) forces acting on the system.

 • Update the momentum of the system, using the Momentum Principle: .

 • Update the position: .

 • Repeat.

Here is how the Momentum Principle can be translated into VPython:

• **Inside the loop, create a new vector variable named F\_air, and assign it the value** **, using the appropriate VPython syntax. (Look at how you created a variable to represent the momentum vector.)**

• **After calculating the net force, use the Momentum Principle to update the momentum.**

• **After updating the momentum, use the new momentum to update the position.**

By experimentation, determine a value for F\_air, the force by the air on the fancart, that produces the following motion:

**The cart starts at the right end of the track and moves to the left, gradually slowing down, and coming to a stop near the left end of the track; it turns around and moves to the right, speeding up.** (Note that you need to let the loop run longer in order to see this behavior.)

**6** **Final Challenge**

Explore changing something about the fancart program to achieve some new kind of motion. So far in this program, we've been restricting the cart's motion to 1D. Can you think of something else to try?

 **CHECKPOINT: Ask the Instructor to look over your program and visual output.**

**7** **Document your work.**

Post screen shots of your code and the resulting display windows to your blog for your cart moving back and forth. Investigate using camstudio or other screen capture tools to create a video of the animation of your cart for your blog.

Calculating and Displaying Gravitational Force

**OBJECTIVES**

In a future program you will model the motion of a spacecraft traveling around a planet and a moon, which will require repeated calculation of gravitational forces.

The objectives of the current exercise are:

• to learn how to instruct VPython to calculate the vector gravitational force on an object due to another object;

• to write the instructions in a symbolic form that can later be used in an iterative calculation to predict the motion of the spacecraft even though the gravitational force is changing in magnitude and direction; and

• to create and scale arrows to represent the gravitational force on an object.

**TIME**

You should plan to finish this activity in 40 minutes or less.

**PLANNING**

On a whiteboard draw a diagram like the one below. Each numbered location represents the position of a different spacecraft. (A single spacecraft near a planet would not move in a straight line.)

• At each numbered location, draw an arrow representing the gravitational force on a spacecraft at that location, due to the planet.

• **Make sure the direction of your arrows is correct, and that the length of the arrow is proportional to the magnitude of the quantity it represents.**

**Look at your diagram. Does it make sense? Compare your work to that of another group.**

**REVIEW: COMPUTER PROGRAM ORGANIZATION**

A computer program consists of a sequence of instructions.

The computer carries out the instructions one by one, in the order in which they appear, and stops when it reaches the end.

Each instruction must be entered exactly correctly (as if it were an instruction to your calculator).

If the computer encounters an error in an instruction (such as a typing error), it will stop running and print a red error message.

**1** **Setup statements**

**Using VIDLE for VPython, create a new file and save it to your own space. Make sure to add ``.py'' to the file name.**

Enter the following statement in the editor window:

**from visual import \***

Recall that if you are using a version of Python earlier than 3.0, in order for 1/2 to mean 0.5, you need to place the following statement as the first statement in your program, before the import of visual:

**from \_\_future\_\_ import division**

You don't need this statement if you are using Python 3.0 or later, but it doesn't hurt, because it is simply ignored by later versions of Python.

**2** **Constants**

Since you will be calculating a gravitational force, you will need the constant  (without units):

**G = 6.7e-11**

You can also put the masses of the planet and the spacecraft in this section. Define constants to represent:

• the mass of the spacecraft (15e3 kg) (you could call this ``**mcraft**'', for example)

• the mass of the planet (6e24 kg) (you might call it ``**mplanet**'')

**3** **Creating an object**

* **Create a sphere object located at the origin to represent the planet, and call it ``planet''. Its radius should be 6.4e6 m, and its color should be something other than white.**
* **Create a second sphere object named ``craft'' to represent a spacecraft at location**  **m. You will need to exaggerate the radius of the craft to make it visible; try 3e6 m. Make its color different from the color of the planet.**
* **Run the program** by pressing F5. Arrange your windows so the Python Shell window is always visible.
* **Kill the program** by closing the graphic display window (or by rerunning your program).

**4** **Calculations**

In a program that models the motion of objects, calculations that are to be repeated are placed inside a loop. In the current exercise we will do a calculation only five times, so it isn't really necessary to use a loop -- once you get the first calculation right, you can copy and paste to do the others. This is inelegant, but acceptable for this exercise. (You can however use a loop if you prefer.)

**5** **Gravitational force law**

A sphere of mass  attracts a sphere of mass  with a (vector) force , where

 • 

 •  is a relative position vector pointing from object 1 toward object 2 (``final minus initial'')

 •  (pronounced ``r-hat'') is a unit vector pointing from object 1 toward object 2

The steps in calculating gravitational force in VPython are the same as the steps you use on paper:

 1. Calculate the relative position vector  that points from the planet toward the spacecraft

 2. Calculate its magnitude 

 3. Use  to calculate the magnitude of the gravitational force 

 4. Calculate the unit vector , which points from object 1 (planet) toward object 2 (spacecraft)

 5. Calculate the vector gravitational force acting on the spacecraft, the product of its magnitude and direction: 

**All of the instructions you type should use only symbolic quantities: No numbers (except for exponents, etc.)**

**Now translate the algebraic expressions into VPython expressions.**

1. *The relative position vector*

You know the vector positions of the two objects, which are **craft.pos** and **planet.pos**.

* **Add a statement to your program to calculate a vector r that points from the planet to the spacecraft, representing the vector . Think about what you know about calculating relative position vectors between two objects.** *Don't use any numbers, just symbols. The point is for VPython to do the numerical calculations for a variety of positions of the planet and the spacecraft.*

r = ?

2. *The magnitude of the relative position vector*

In order to instruct the computer to calculate the magnitude  of the relative position vector, you need to know the following:

* **The components of a vector in VPython are given by its x, y, and z attributes. For example, craft.pos.x is the x component of the position of the spacecraft, and r.y is the y component of the vector r that you created.**
* **To calculate the square of a quantity, use two asterisks.**

For example, 3\*\*2 gives 9.

* **To calculate the square root of a quantity, use the function sqrt.**

For example, sqrt(9) is 3.

Knowing these features of VPython,

* **Add a statement to your program to calculate the magnitude  of the relative position vector:**

**rmag = ?**

3. *The magnitude of the gravitational force*

Using the quantity  that you just calculated (**rmag**), and the masses **mcraft** and **mplanet** (or whatever you chose to call them),

* **Add a statement to your program to calculate the magnitude  of the gravitational force. You should define G near the start of your program. Use this symbol G rather than a number in calculating :**

Fmag = ?

4. *The unit vector*

* **You know both the vector  (r) and its magnitude (rmag). Use these quantities to add a statement to your program to calculate the unit vector  (pronounced ``r-hat''):**

**rhat = ?**

5. *The gravitational force as a vector*

Now you have everything you need to be able to calculate the gravitational force that the planet exerts on the spacecraft: 

* **Add a statement to your program to calculate the net vector force acting on the spacecraft. We'll assume that the planet is the only object near enough to have a significant effect on the spacecraft.**

**Fnet = ?**

* **Add a print statement to your program to show the components of the net force:**

print(“Fnet =”, Fnet)

* **Run the program and ask yourself whether the signs of the force components makes sense (then make changes to your program if necessary).**

**Visualizing the force vector with an arrow**

Having calculated the gravitational force vector, we want to visualize it by displaying an arrow representing the vector.

• **Add a statement to your program to create a yellow arrow on the screen that represents the gravitational force acting on the spacecraft (object 2). Choose pos and axis attributes so that the tail of the arrow is on the spacecraft and the arrow points toward the planet. *Do NOT use numbers*! Write the values for the pos and axis attributes symbolically, in terms of the quantities you have already calculated.**

If you have calculated the gravitational force correctly, you probably don't see an arrow! The force is so small you have to scale it up to be able to see it, by multiplying by a scalar factor. How do you pick a scale factor?

**6** **Scalefactors**

Watch **VPython Instructional Videos: 5. Scalefactors**

<http://www.youtube.com/VPythonVideos> Use what you have learned from this video to determine an appropriate scalefactor for this force arrow. Remember to define the scalefactor in the ##Constants section and use it later in the program where the force arrow is defined. The same scalefactor should be used for all force arrows.

**The force at other positions**

In order to see how the force on the spacecraft would vary at different positions, add code to create 4 other spacecraft (spheres) at the locations listed below, and to calculate and display the force on each spacecraft. You can simply make four copies of your existing code and change the spacecraft position in each copy, or you could put your existing code inside a loop.

 m

 m

 m

 m

• **Run your program so you see your display showing the spacecraft and the planet.**

• **If the arrows get too big, reduce your scalar Fscale so that the arrows don't run into each other.**

• **You must use the *same* scalefactor with *all* of your arrows, so that all the force vectors are consistent.**

**Look at your display. Does it make sense? Does it look like the diagram you drew on the whiteboard? Compare your result to that of another group.**

**Be sure to make screen shots of your final image for inclusion in your blog.**

**Optional additions**

***Label the arrows:*** Look in the online VPython manual to find out how to add a ``label'' to your display. Label the spacecraft and the planet. Also label at least one of the arrows representing force. To access the VPython manual, pull down the Help menu and select ``Visual''.

***Write a loop:*** In order to emphasize the physics rather than programming techniques, it was acceptable to make multiple copies of the calculational statements, which is clunky, and not feasible if there are hundreds of positions involved. It is not difficult to have just one copy of the calculations in a while loop where you increment the x component of the position of the spacecraft each time through the loop. Try it!

***Place the spacecraft with the mouse:*** The following statement waits for you to click the mouse somewhere on the screen and gives you the location of the mouse (``scene'' is the name of the default graphics window):

location = scene.mouse.getclick()

With this statement in a loop, you could use this mouse location to specify a spacecraft's position and then calculate and display the gravitational force on the spacecraft at that location, and then repeat.

You can read more about handling mouse events, including continuous dragging, in the VPython documentation (``Visual'' on the Help menu). Also there are a number of example programs that use the mouse which you can find by choosing File  Open immediately upon starting up VIDLE (on a Mac, go to /Applications/VPython-PyX.Y/examples).

A Spacecraft Voyage, Part 1: Spacecraft and Earth

**OBJECTIVES**

In this program you will model the motion of a spacecraft. You will use your working program to explore the effect of the spacecraft's initial velocity on its trajectory.

**TIME**

You should finish this activity in 45 minutes or less.

**1** **Explain and Predict Program Visualization**

 A minimal working program has been provided below.

**DO NOT RUN THE PROGRAM YET. Read through the program together. Make sure everyone in the group understands the function of *each* program statement. Reading and explaining program code is an important part of learning to create and modify computational models.**

from \_\_future\_\_ import division

from visual import \*

scene.width =800

scene.height = 800

#CONSTANTS

G = 6.7e-11

mEarth = 6e24

mcraft = 15e3

deltat = 60

#OBJECTS AND INITIAL VALUES

Earth = sphere(pos=vector(0,0,0), radius=6.4e6, color=color.cyan)

craft = sphere(pos=vector(-10\*Earth.radius, 0,0), radius=1e6, color=color.yellow)

trail = curve(color=craft.color) ## for leaving a trail behind the craft

vcraft = vector(0,2e3,0)

pcraft = mcraft\*vcraft

t = 0

#CALCULATION LOOP: ALL REPEATED CALCULATIONS GO INSIDE THE LOOP

while t < 10\*365\*24\*60\*60:

 rate(100) ## slow down motion to make animation look nicer

 craft.pos = craft.pos + (pcraft/mcraft)\*deltat

 trail.append(pos=craft.pos) ## this leaves a trail behind the spacecraft

 t = t+deltat

After reading *every* line of the program, answer the following questions:

• What is the physical system being modeled? In the real world, how should this system behave? On the left side of the whiteboard, draw a sketch showing how you think the objects should move in the real world.

• Will the program *as it is now written* accurately model the real system? **DO NOT RUN THE PROGRAM YET.** Study the program again. On the right side of the whiteboard, draw a sketch of how the objects created in the program will move on the screen, based on your interpretation of the code.

• Run your program *after* everyone agrees on both predictions. Discuss how closely your prediction of what the program would do matches what you see in the visual output.

**2** **Run the program**

• How did your prediction compare to what you saw? Did something happen that you didn't expect to happen?

**3** **Modify the program**

• **What calculations are needed to express the interactions between the spacecraft and the Earth?** Recall the previous discussion of iterative prediction of motion. All repeated calculations go inside the loop in the program.

 - Calculate the (vector) forces acting on the system and their sum, .

 - Update the momentum of the system: .

 - Update the position: .

 - Repeat. ->(0.2,0)(-.3,0)(-.3,2.0)(0.2,2.0)

• **Modify the program with these calculations. Where should you place these new lines of code in the program?** The previous two programs you have written may be helpful resources.

• **Use symbolic names defined in the program. You may need to add other symbolic names in your calculations.**

• **Finally, run the program. Is the behavior what you expected? If not, check that your code correctly expresses the relevant physics, and that you have placed repeated calculations inside the while loop.**

**4** **Visualize the momentum vector with an arrow**

In a previous program, you used arrows to visualize gravitational force vectors. In this program, you will use an arrow to visualize the momentum of the spacecraft.

You will create an **arrow** before the **while** loop and update its position and axis inside the loop, as the spacecraft's momentum changes.

**It is very important NOT to create the arrow inside the loop, because then you would have many arrows rather than one. Just as you create ONE sphere to represent the spacecraft, and then inside the loop you continually modify the sphere's position, so also you create ONE arrow to represent the momentum, and then inside the loop you continually modify the arrow's position and axis.**

You need to know the approximate magnitude of the momentum in order to be able to scale the arrow to fit into the scene, so just before the loop, print the momentum vector:

**print('p=', pcraft)**

In the **#CONSTANTS** section of your program, add this scale factor

**pscale = ??**

Also insert the following statement **in the #OBJECTS section of your program, NOT inside the loop**, because that would create a very large number of arrows:

**parr = arrow(color=color.green)**

This statement creates an arrow object with default pos and axis attributes.

• **Comment out any print statements that are inside your loop, because they slow the program down.**

• **Inside your loop, update the pos attribute of the parr arrow object to be at the center of the spacecraft, and on a separate line update its axis attribute to represent the current vector value of the momentum of the spacecraft (multiplied by pscale).**

parr.pos = ?

parr.axis = ?

• **You may have to adjust the scale factor once you have seen the full orbit.**

**5** **Answer questions about changes in the spacecraft's momentum**

• **For this elliptical orbit, what is the direction of the spacecraft's momentum vector? Tangential? Radial? Something else?**

• **What happens to the momentum as the spacecraft moves away from the Earth?**

• **As it moves toward the Earth?**

• **Why? Explain these changes in momentum in terms of the Momentum Principle.**

• **Compare your answers to those of another group.**

**6** **Answer questions about the effect of initial velocity on the motion**

• **Approximately, what minimum initial speed is required so that the spacecraft ``escapes'' and never comes back? You may have to zoom out to see whether the spacecraft shows signs of coming back. You may also have to extend the time in the while statement.**

• **What initial speed is required to make a nearly circular orbit around the Earth? You may wish to zoom out to examine the orbit more closely.**

• **Why does changing the initial velocity have an effect on the orbit?**

**7** **Optional: Detecting a collision**

If your spacecraft collides with the Earth, the program should stop. Add code similar to the following inside your loop (using the name you defined for the distance between the spacecraft and the center of the Earth):

  **if rmag  Earth.radius:**

 **break**

This code tells VPython to get out of the loop if the spacecraft touches the Earth.

**8** **Report your work on your blog**

Include answers to each of the questions plus a video of your ship in motion about the earth.

A Spacecraft Voyage, Part 2:

From the Earth to the Moon

**OBJECTIVES**

In this program you will model the motion of a spacecraft traveling from the Earth to the Moon. To do this you will iteratively (repeatedly):

 • calculate the gravitational force on the spacecraft by the Earth and the Moon

 • apply the Momentum Principle to update the momentum of the spacecraft

 • update the position of the spacecraft

You will use your working program to

 • explore the effect of the spacecraft's initial velocity on its trajectory

 • study the effect of your choice of  (deltat) on the accuracy of your predictions

This activity is a practical example of the use of a computer program to do repeated calculations. This problem (three gravitationally interacting objects) cannot be solved any other way -- it is possible to write down a set of calculus equations, but they will not have a general symbolic solution!

**Resources you will find useful:**

 • Start from a copy of your program to predict the motion of a spacecraft due just to the Earth

 • The VPython reference manual (Help menu, choose Visual)

**Organization of these instructions:**

The general goal of each section will be stated at the beginning of the section. Hints and detailed suggestions for implementation are given in following sub-sections.

**TIME**

You should finish this part of the lab in 45 minutes or less.

**Data**

|  |  |
| --- | --- |
|  Mass of Earth: 6e24 kg  |  Radius of Earth: 6.4e6 m |
|  Mass of spacecraft: 15e3 kg  |  Radius of spacecraft: very small (exaggerated in program) |
|  Mass of Moon: 7e22 kg  |  Radius of Moon: 1.75e6 m |
|  Distance from Earth to Moon: 4e8 m  |  G=6.7e-11 N  m/kg |

**1** **Including the effect of the Moon**

In the real world, the net force on a spacecraft is rarely due to only one other object. By adding the Moon to your program, you will get to observe the kind of complex motion that the interaction of three or more objects can produce.

Create spheres to represent the spacecraft and the Moon, in the given initial positions:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  **Object Name**  |  **Kind of Object**  |  **Initial Location**  |  **Radius**  |  **Color** |
|  **spacecraft**  |  **sphere**  |  10 Earth radii to left of the Earth  |  1e6 m  |  white |
|  **Moon**  |  **sphere**  |  4e8 m to the right of the Earth  |  1.75e6 m  |  white |



Inside your **while** loop, add a calculation of the gravitational force that the Moon exerts on the spacecraft, and calculate the net force due to the force of the Earth and the force of the Moon. Use the net force to update the momentum of the spacecraft.

* **Inside your while loop, add a calculation of the gravitational force that the Moon exerts on the spacecraft.**
* **Calculate the *net* force on the spacecraft, and use the *net* force to update the momentum.**
* **Add a check for crashing on the Moon, like your check for crashing on the Earth.**
* **Lengthen the loop time to 60 days, to follow the more complicated orbits that can occur.**
* **Initially, use a step size of 10 seconds.**
* **Find an initial speed that leads to crashing on the Moon.**
* **Find an initial speed that yields a ``figure-8" orbit that loops around the Moon before returning to Earth. Make a note of it as a comment in your program.**
* **How sensitive is this to the initial velocity? How much can you change the initial speed and still get a figure-8 orbit?**
* **Play around with the initial speed and see what other kinds of orbits you can find.**

When the spacecraft interacted solely with the Earth (a ``two-body" interaction), there were only a few kinds of orbits possible, and they were quite simple curves (circle, ellipse, parabola, hyperbola, straight line). Small changes in the initial velocity typically led to small changes in the orbit. For example, an ellipse merely became a longer ellipse with a larger initial speed.

In the ``three-body" interaction of Earth, Moon, and spacecraft, there is a much larger variety of orbits. If just three bodies show complex behavior, no wonder a macroscopic system such as you is highly complex, since you contain an astronomical number of interacting atoms! Also, an extremely slight change in the initial velocity can make a major change in the motion: the orbit is highly sensitive to the initial velocity. The rich variety of types of motion, and the high sensitivity to initial velocity, are typical of complex systems. This is the subject of the relatively new science of ``chaos" which you can read more about in James Gleicks “Chaos: Making a New Science”

**2** **Answer these questions about a particularly interesting trajectory**

• **Give the spacecraft a speed of 3.27 km/s (3.27e3 m/s), headed in the +y direction.**

What happens?

• **Why does coming nearly to a stop lead to retracing the path?**

Explain in terms of the Momentum Principle and the gravitational force law.

• **Discuss your results and answers with those of a nearby group.**

**3** **Accuracy**

If you use a very large , the calculation is inaccurate, because during that large time interval the force changes a lot, making the momentum update inaccurate, and the momentum changes a lot, making the position update inaccurate. On the other hand, if you use a very small  for high accuracy, the program runs very slowly. In computational modeling, there is a trade-off between accuracy and speed.

How can you tell whether an orbit is accurate? There's a simple test: **Make  smaller and see whether the motion changes.** That is, see whether the orbit changes shape. (Obviously the program will run more slowly).

• You've been using a step size of 10 seconds. Try a step size one-fifth as large (2 seconds). With an initial speed of 3.27 km/s, is the orbit the same? If the orbit is the same using this smaller step size, that implies that a step size less than or equal to 10 seconds is adequately short to give accurate results.

• To see the effects of  being too large, make  100 times as large as you had been using (1000 seconds).

• Your instructor may ask you to demonstrate an accurate orbit and a deliberately inaccurate orbit.

**4** **Answer these questions about the effect of step size on accuracy**

• **Does the 10 second step size give an accurate orbit? How do you know?**

• ***Why* does the large step size (1000 seconds) give an inaccurate orbit?**

• **Discuss your answer with those of a nearby group.**

**After checking with another group, ask the instructor to check your work.**

**Before turning in your program, restore it to the following state:**

**10 second step size**

**initial speed of spacecraft 3.27e3 m/s in the +y direction**

**Be sure to answer all of the questions in your blog and include an animation of your ship orbiting the earth.**

**5** **Playing around**

Shoot the spacecraft straight toward the Moon from near the surface of the Earth, starting at location , with an initial speed of 11 km/s. What happens? Why? What happens if you increase or decrease the initial speed? It's interesting to watch the momentum vector and think about the effects of the forces on the momentum during this motion.

You can make a more realistic model if you let the Moon and Earth orbit each other while the spacecraft is moving. Calculate all the forces among the three bodies, then update all of the momenta before updating any of the positions. That way you calculate the forces in a consistent way at a particular instant. You will have to experiment a bit with the initial velocity to get the nearly circular orbit of the Moon around the Earth (the Earth follows a nearly circular orbit, too, but it is only a small wobble because the Earth's mass is much larger than the Moon's mass.)

You might even try to model the Sun-Earth-Moon system, with or without the spacecraft. A practical difficulty with visualizing the Sun-Earth-Moon system is that the Earth-Moon system is very small compared to the great distance to the Sun, so it's hard to see the details of the Moon's motion. However, you can continually update the center of the scene by resetting **scene.center** inside the loop to follow the Earth-Moon system, and zoom in on this region. Alternatively, you can set **scene.autocenter = True**, which makes the centering automatic.

A Spacecraft Voyage, Part 3

Kinetic and Potential Energy

**OBJECTIVES**

In this activity you will analyze the flow of energy between kinetic energy and potential energy in a multiparticle system. You will

• calculate and graph the kinetic energy, potential energy, and the sum of  as a function of time for the system of a spacecraft and the Earth (at first, without the influence of the Moon)

• calculate and graph , , and  vs. time for the system of the spacecraft, Earth, and Moon

• graph , , and  as a function of separation of the spacecraft from the Earth

• study the effect of your choice of  (deltat) on the accuracy of your predictions

**Resources you will find useful:**

Start from a copy of your program to predict the motion of a spacecraft interacting with the Earth and Moon

**TIME**

You should finish this activity in 40 minutes or less.

On Your Whiteboard

**1** **Predictions: Spacecraft and Earth only (no Moon)**

Draw graphs of kinetic energy  (in green), potential energy  (in red), and  (in blue) vs. time (not separation!) for the system of spacecraft + Earth, for the following situations (not including the Moon):

• Spacecraft orbits the Earth in an elliptical orbit

• Spacecraft orbits the Earth in a circular orbit

• Spacecraft heads straight away from the Earth and does not return

**Compare your predictions to those of a nearby group, then ask a INSTRUCTOR to check your predictions.**

**2** **Adding graphs to your program: Energy vs. time**

• Immediately after the **from visual import\***, add the following two lines:

**from visual.graph import\*** **## invoke graphing routines**

**scene.y = 400** **## place graph 400 pixels below top of screen**

• Just before the while loop, add the following three lines:

**Kgraph = gcurve(color=color.yellow)**  **## create a gcurve for kinetic energy**

**Ugraph = gcurve(color=color.red)**  **## create a gcurve for potential energy**

**KplusUgraph = gcurve(color=color.cyan)**  **## create a gcurve for sum of K+U**

• Inside the while loop, at the end, calculate and plot kinetic energy, potential energy, and the sum of kinetic and potential energy as a function of time. Omit the Earth-Moon potential energy term, which does not change, but include all others:

**K = ## you must complete this line**

**U = ## you must complete this; omit the constant term for the Earth-Moon interaction**

**Kgraph.plot(pos=(t,K))**  **## add a point to the kinetic energy graph**

**Ugraph.plot(pos=(t,U))**  **## add a point to the potential energy graph**

**KplusUgraph.plot(pos=(t,K+U))**  **## add a point to the K+U graph**

**3** **Testing your program: A zero mass Moon**

* In order to compare your predictions to your results, set the mass of the Moon to zero. This will have the same effect as temporarily removing the Moon from your program, without having to change any code.
* Set the initial velocity of the spacecraft to a velocity that produces an elliptical orbit.
* Run your program.
* Compare your graphs to your predictions. Are your graphs correct? How can you tell?

**4** **Additional questions to answer**

* Experiment with the size of the time step deltat. How large can you make deltat before the calculated values become inconsistent with the Energy Principle? What do you observe in your graphs that indicates that deltat is too large?
* Plot energy vs. separation instead of time. Are your graphs correct? It is useful in your gcurve specification to say ``dot=True'' in which case a moving dot is plotted on the graph, so that you can see where the plotting is taking place on a repetitive graph (VPython 5.3 or later; you can determine the version with ``print(version)'').
* Revert to plotting energy vs. time, and set the mass of the Moon back to the correct mass. Set the initial velocity of the spacecraft to a velocity that produces a figure-eight type path. Make sure you calculate all energy terms, except that you may omit the potential energy due to the Earth-Moon interaction, which is large and constant, and makes it difficult to see other effects. You can drag the mouse over the graph to see crosshairs that let you read values accurately off the graph (VPython 5.61 or later).
* Explain the resulting graphs.

**Compare your results and explanations to those of a nearby group, then ask a Instructor to check your predictions.**

**Post your program to your blog. The program you submit should produce energy vs. time graphs for the spacecraft-Earth-Moon system, for a figure-8 type path. Your blog entry should include answers to the questions and images of both the figure-8 path and the graphs you generated.**

Computer Model of Spring-Mass System

**1** **Explain and Predict Program Visualization**

A minimal working program has been provided below. **DO NOT RUN THE PROGRAM YET. Read through the program together. Make sure everyone in the group understands the function of *each* program statement. Reading and explaining program code is an important part of learning to create and modify computational models.**

from \_\_future\_\_ import division

from visual import \*

scene.width=600

scene.height = 760

## constants and data

g = 9.8

mball = 1 ## change this to the appropriate mass (in kg) from your mass-spring experiment.

L0 = 0.26 ## this is an approximate relaxed length of your spring in meters measured in lab.

ks = 1 ## change this to the spring stiffness you measured (in N/m)

deltat = .01

t = 0 ## start counting time at zero

## objects

ceiling = box(pos=(0,0,0), size = (0.2, 0.01, 0.2)) ## origin is at ceiling

ball = sphere(pos=(0,-0.3,0), radius=0.025, color=color.orange) ## note: spring compressed

spring = helix(pos=ceiling.pos,axis=ball.pos-ceiling.pos,color=color.cyan, thickness=.003 , coils=40, radius=0.015) ## change the color to be your spring color

trail = curve(color=ball.color) ## for leaving a trail behind the ball

## initial values

pball = mball\*vector(0,0,0)

Fgrav = mball\*g\*vector(0,-1,0)

## improve the display

scene.autoscale = False ## don't let camera zoom in and out as ball moves

scene.center = vector(0,-L0,0) ## move camera down to improve display visibility

scene.mouse.getclick() ## Animation doesn't start when you Run Program.

## calculation loop

while t < 30:

 rate(100)

 Fnet = Fgrav

 pball = pball + Fnet\*deltat

 ball.pos = ball.pos + pball/mball\*deltat

 spring.axis = ball.pos - ceiling.pos

 trail.append(pos=ball.pos) ## this adds the new position of the ball to the trail

 t = t + deltat

After reading *every* line of the program, answer the following questions:

 • What is the physical system being modeled? In the real world, how should this system behave? On the left side of the whiteboard, draw a sketch showing how you think the objects should move in the real world.

 • Will the program *as it is now written* accurately model the real system? **DO NOT RUN THE PROGRAM YET.** Study the program again. On the right side of the whiteboard, draw a sketch of how the objects created in the program will move on the screen, based on your interpretation of the code.

• Run your program *after* everyone agrees on both predictions. Discuss how closely your prediction of what the program would do matches what you see in the visual output.

**2** **Extend Program to Include Additional Interactions**

**Change the values in the program for the ball's mass and the spring stiffness using the information you obtained from your previous experiments with a mass and spring.**

Modify the program to model what happens when you connect the ball to a spring, and release it from rest.

 • All repeated calculations go inside the calculation loop in the program:

 - Calculate the (vector) forces acting on the system and their sum, .

 - Update the momentum of the system: .

 - Update the position: .

 - Repeat.

 • Remember that the spring force can be written as , where , and  goes from the fixed end of the spring to the other end.

• Use the attributes of 3D objects and other variables or constants that already exist in the program you obtained.

• Predict what you would expect to see when you run the program.

• Run the program and make any necessary changes to the code to achieve your goal. Is the behavior what you expected? If not, check that your code correctly expresses the relevant physics.

**Describe to how you can re-create what happens in this computer model with the mass-spring lab equipment at your station.**

**3** **Using a Graph to Determine the Period**

Have your program produce a graph of the y-coordinate of the ball's position vs. time. Use this graph to determine the period of the oscillating system in your computer model. Here are reminders on how to do this:

After ``from visual import \*'' add these statements:

from visual.graph import \* # import graphing module

scene.y = 400 # move scene below graph

In the ##objects section of your program, create a gcurve object, for plotting the position of your ball:

ygraph = gcurve(color=color.yellow)

Inside the loop, after updating the momentum and position of the ball, and the time, add the following statement:

ygraph.plot(pos=(t, ball.pos.y))

**Answer the following questions:**

* Why doesn't the graph cross zero?
* What is the period of the oscillations shown on the graph? (You will find it easier to read the period off the graph if you change the while : statement to make the program quit after a small number of oscillations. Also, with VPython 5.61 or later you can use the mouse to drag crosshairs on a graph. You can determine the version with ``print(version)''.)
* How does the period of your model system compare with the period you measured for your real system?
* What does the analytical solution for a spring-mass oscillator predict for the period?
* Should these numbers be the same? If they are not, why not?
* Make the mass 4 times bigger. What is the new period? Does this agree with theory? (Afterwards, reset the mass to its original value.)
* Make the spring stiffness 4 times bigger. What is the new period? Does this agree with theory? (Afterwards, reset the spring stiffness to its original value.)
* Make the amplitude twice as big. (How?) What is the new period? Does this agree with theory? (Afterwards, reset the amplitude to its original value.)
* Record your answers to these questions as comments in your program.

**Compare your answers and graph to those of another group. Be sure to post your program, graph and video to your blog.**

Computer Model of Spring-Mass System

Part 2

**QUESTIONS:**

Q1: How does energy flow within the system?

Q2: What initial conditions must you specify in your program in order to get your virtual mass-spring system to oscillate in all three dimensions, instead of just staying in one plane?

**1** **Add Graphs of Energy vs. Time**

Start from the spring-mass program you wrote in a previous assignment.

1. Comment out the  vs. time graph defined in the ##OBJECTS section and in the loop.
2. Calculate *K* and *U* for the system consisting of only the Mass and the Spring.
3. Make graphs of *K*, *U*, and *K+U* vs. time for the system (Mass+Spring).

• Just before the while loop, add the following three lines:

**Kgraph = gcurve(color=color.yellow)**  **# create a gcurve for kinetic energy**

**Ugraph = gcurve(color=color.red)**  **# create a gcurve for potential energy**

**KplusUgraph = gcurve(color=color.cyan)**  **# create a gcurve for sum of K+U**

• Inside the while loop, at the end, calculate and plot kinetic energy, potential energy, and the sum of kinetic and potential energy as a function of time for the **Mass + Spring** system:

**K = # complete this line**

**U = # complete this line**

**Kgraph.plot(pos=(t,K))**  **# add a point to the kinetic energy graph**

**Ugraph.plot(pos=(t,U))**  **# add a point to the potential energy graph**

**KplusUgraph.plot(pos=(t,K+U))**  **# add a point to the K+U graph**

Should  be constant for this system of Mass + Spring? Why or why not?

What do you observe when you run the program?

For what choice of system would you expect  to be constant?

Revise your program to display graphs of energy for such a system.

What do you observe when you run the program?

**Compare your answers and graph to those of another group.**

Show your running program to an Instructor and explain your answers.

**2** **3D Motion**

1. Make sure that your program still leaves a trail behind the moving mass, as was true in your original spring-mass program.

2. Find initial conditions that produce oscillations not confined to a single plane. Zoom and rotate to make sure the oscillations are not planar. Observe the energy graphs.

**Submit the answers to the questions previously, a program listing and an animation of your program creating a non-planar orbit, with  constant.**

**3** **Just for Fun**

True Stereo (just for fun): Get a pair of red-cyan glasses from your Instructor. Add the following statement near the beginning of your program:

 scene.stereo = "redcyan"

Run your program and look through the glasses (red over left eye).

Calculating and displaying the electric field of a single charged particle

**1** **Objective**

Using your calculator, you have calculated the electric field at an observation location due to a single charged particle. The somewhat tedious process of calculating electric field vectors can be automated by programming a computer to do this. Once we have written the instructions for calculating the electric field at one observation location, we can re-use the instructions to allow us to calculate and display the electric field in multiple locations, so we can examine the 3D pattern of field created by a charged particle.

In this activity you will write VPython instructions to calculate and display the electric field due to a single point charge, at several observation locations. We start with a single point charge because you know what the pattern of electric field should look like, so you can make sure your approach is correct.

**2** **Whiteboard Work**

**2.1** **Predict the Field Pattern**

**Consider the diagram below. If a proton were placed at the location shown by the red sphere, what would you expect the electric field to be at the observation locations marked by colored boxes? (The cyan boxes lie in the  plane and the magenta boxes lie in the  plane.) Draw a diagram on a whiteboard showing your prediction.**

**2.2** **Electric Field Equation**

**On your whiteboard, write the vector equation for the electric field of a point charge. Make sure every detail of the equation is correct.**

**2.3** **Calculation Steps**

**On your whiteboard, briefly outline the steps involved in calculating the electric field, as if you were reminding a friend how to do it. For example, the first step might be:**

 1. Find the relative position vector  from the source location to the observation location

 2. ...

 3. ...

 4. ...

 5. ...

 **Checkpoint: Share your work with other students.**

**3** **Read and Explain a Minimal Working Program**

A minimal working program is shown in the space below.

from \_\_future\_\_ import division

from visual import \*

## CONSTANTS

oofpez = 9e9 ## OneOverFourPiEpsilonZero

qproton = 1.6e-19

## OBJECTS

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

xaxis = cylinder(pos=(-5e-10,0,0), axis=vector(10e-10,0,0),radius=.2e-11)

yaxis = cylinder(pos=(0,-5e-10,0), axis=vector(0,10e-10,0),radius=.2e-11)

zaxis = cylinder(pos=(0,0,-5e-10), axis=vector(0,0,10e-10),radius=.2e-11)

## the position of the arrow is the observation location:

Earrow1 = arrow(pos=vector(3.1e-10,-2.1e-10,0), axis = vector(1e-10,0,0), color=color.orange)

## CALCULATIONS

## write instructions below to tell the computer how to calculate the correct

## electric field E1 at the observation location (the position of Earrow1):

## change the axis of Earrow1 to point in the direction of the electric field at that location

## and scale it so it looks reasonable

## additional observation locations; do the same thing for each one

Earrow2 = arrow(pos=vector(3.1e-10,2.1e-10,0), axis = vector(1e-10,0,0), color=color.orange)

Earrow3 = arrow(pos=vector(-1.1e-10,-2.1e-10,0), axis = vector(1e-10,0,0), color=color.orange)

Earrow4 = arrow(pos=vector(-1.1e-10,2.1e-10,0), axis = vector(1e-10,0,0), color=color.orange)

Earrow5 = arrow(pos=vector(1e-10,0,3e-10), axis = vector(1e-10,0,0), color=color.orange)

Earrow6 = arrow(pos=vector(1e-10,0,-3e-10), axis = vector(1e-10,0,0), color=color.orange)

**Read the program carefully, line by line, starting at the beginning. Before modifying the program, answer the following two questions:**

 1. Program organization: The minimal working program is organized into several sections. What is the purpose of each section?

 2. Predict program output: Do you think the display generated by the minimal program will match the field pattern you predicted? Why or why not? Be prepared to explain your reasoning to your instructor.

**Navigate to** [**http://profmason.com/?page\_id=1824**](http://profmason.com/?page_id=1824) **,copy and paste it into a blank python file and save it. Remember to give it the extension ``.py''**

**Run the minimal program: Run the program and compare what you see to your original whiteboard prediction of the field pattern. Rotate the display to help understand where objects are located.**

**4** **Modify the Minimal Program**

**4.1** **Calculating the Electric Field at One Location**

 •  **At the appropriate location in the program, insert VPython instructions to tell the computer how to calculate the electric field at the first observation location. Your instructions should follow the outline you wrote on the whiteboard. Be sure to use symbolic names.**

**4.1.1** **Useful VPython functions**

- The components of a vector may be referred to as , , and  For example,the  component of a vector named  **rvector would be rvector.y**

- To square a quantity use two asterisks.  would be written  **x\*\*2 .**

- To calculate the magnitude of a vector you can either use  **sqrt(r.x\*\*2 + r.y\*\*2 + r.z\*\*2) or you can simply use mag(r) . (The online help for Visual has a detailed list of all vector operations; see *Vector Operations*).**

•  **Print the value of the electric field at the first observation location: print(E)**

**4.2** **Representing the Electric Field by an Arrow**

In order to use an arrow to visualize the electric field at one location, it is necessary to set both the direction and the magnitude of the arrow's axis.

**4.2.1** **Changing Attributes of Objects**

Any attribute of an object may be changed after the object is created. For example, suppose you have created an arrow:

 **Earrow1 = arrow(pos=vector(1,2,3), axis=vector(0,0,3e-10), color=color.red)**

Later in the program you could change the color of the arrow with an instruction like this:

 **Earrow1.color = color.orange**

You could change the axis of the arrow with an instruction like this (assume that **Fnet** is a vector quantity you have calculated)**:**

 **Earrow1.axis = Fnet**

**4.3** **Orienting and Scaling your Arrow**

•  **Change the axis of the arrow Earrow1 so it points in the direction of the field you calculated, and has a reasonable magnitude. You will need to use a scale factor to make everything visible.**

 **Checkpoint: Ask your instructor to look over your work.**

**5** **Add more observation locations**

•  **Extend your program to calculate and display the electric field at the other 5 observation locations (the locations of the other five arrows). One way to do this is to copy the code you have written to calculate and display the electric field, and paste it in multiple times. This is the way we'll do it this time; later we will learn a more flexible way to do this using a loop. (Optionally, you may use a loop here if you already know how do this quickly.)**

•  **Use the same scalefactor for all arrows**

•  **Print the value of the electric field at each observation location.**

 Make sure everyone in your group thinks your display looks reasonable.

Compare your work to that of a neighboring group.

**Checkpoint: Ask your instructor to look over your work.**

**6** **Document**

Post a copy of your program and a screen shot on your blog. You will get full credit if your program correctly calculates and displays the 6 electric field vectors specified.

**Electric Field of a Dipole**

**1** **Calculating and Displaying the Electric Field**

**1.1** **Objectives**

In this activity you will write a VPython program that calculates (exactly) and displays the electric field of a dipole at many observation locations, including locations not on the dipole axes. These instructions assume that you have already written a program to calculate and display the electric field of a single point charge. To calculate and display the field of a dipole you will need to:

• Instruct the computer to calculate electric field (you may wish to review your work from the previous lab: Calculating and displaying the electric field of a single charged particle)

• Apply the superposition principle to find the field of more than one point charge

• Learn how to create and use a loop to repeat the same calculation at many different observation locations

• Display arrows representing the magnitude and direction of the net field at each observation location

**1.1.1** **Time**

You should finish Part I in 60 minutes or less. Check with your instructor to see whether you are also supposed to do part II.

**1.2** **Program loops**

A program ``loop'' makes it easy to do repetitive calculations, which will be important in calculating and displaying the pattern of electric field near a dipole.

**1.2.1** **Multiple observation locations**

To see the pattern of electric field around a charged particle, your program will calculate the electric field at multiple locations, all lying on a circle of radius  centered on the dipole, in a plane containing the dipole. Instead of copying and pasting code many times, your calculations are carried out in a loop.

Suppose that one of the planes containing the dipole is the  plane. The coordinates of a location on a circle of radius  in the  plane, at an angle  to the -axis, can be determined using direction cosines for the unit vector to that location:





Therefore the observation location is at .

**1.3** **Read and Explain a Minimal Working Program**

A minimal runnable program has been provided in the space below.  **DO NOT RUN THE PROGRAM YET.**

•  **Read the program carefully together, line by line. Make sure everyone in the group understands the function of *each* program statement. Reading and explaining program code is an important part of learning to create and modify computational models.**

•  **After reading *every* line of the program, answer the following questions:**

 - System: What is the physical system being modeled? What is the charge of each particle?

 - System: Approximately what should the pattern of electric field look like?

from \_\_future\_\_ import division

from visual import \*

## constants

oofpez = 9e9 # stands for One Over Four Pi Epsilon-Zero

qe = 1.6e-19 # proton charge

s = 4e-11 # charge separation

R = 3e-10 # display Enet on a circle of radius R

scalefactor = 1.0 # for scaling arrows to represent electric field

## objects

## Represent the two charges of the dipole by red and blue spheres:

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

qplus = qe # charge of positive particle

neg = sphere(pos=vector(-s/2,0,0), radius=1e-11, color=color.blue)

qneg = -qplus # charge of negative particle

## calculations

theta = 0

while theta < 2\*pi:

 rate(2) # tell computer to go through loop slowly

 ## Calculate observation location (tail of arrow) using current value of theta:

 Earrow = arrow(pos=R\*vector(cos(theta),sin(theta),0), axis=vector(1e-10,0,0), color=color.orange)

 ## write instructions below to tell the computer how to calculate the correct

 ## net electric field Enet at the observation location (the position of Earrow):

 ## change the axis of Earrow to point in the direction of the electric field at that location

 ## and scale it so it looks reasonable

 ## Assign a new value to theta

 theta = theta + pi/6

**On the left side of the whiteboard, draw a sketch showing the source charges and how you think the pattern of their electric field should look.**

 - Program: Will the program move one arrow in an animation, or create many arrows?

 - Program: Will the program *as it is now written* accurately model the real system?

**On the right side of the whiteboard, draw a sketch of how the pattern of electric field displayed by the program will look, based on your interpretation of the code.**

 **Checkpoint: Ask your instructor to look over your work.**

**Navigate to** [**http://profmason.com/?page\_id=1824**](http://profmason.com/?page_id=1824) **copy and paste the code into an empty vpython window and save it. Remember to give it the extension ``.py''.**

**Run the program *after* everyone agrees on both predictions. Discuss how closely your prediction of what the program would do matches what you see in the visual output.**

**1.4** **Modify the Minimal Program**

**1.4.1** **Organizing the Calculations**

  **Briefly explain why you can't use this equation: **

It is possible to waste a lot of time writing a program if you do not have a clear outline of the sequence of instructions to guide your work. To avoid this:

**On your whiteboard, briefly outline the steps involved in calculating the net electric field at a single observation location due to the contributions of both of the charges in the dipole. Make your outline detailed, as if you were reminding a friend how to do it. For example, the first step might be:**

1. Find the relative position vector  from the location of the positive charge to the observation location

2. ...

3. ...

**1.4.2** **Writing Code**

•  **At the appropriate location in the program, insert VPython instructions to tell the computer how to calculate the electric field at the first observation location. Your instructions should follow the outline you wrote on the whiteboard. Be sure to use symbolic names.**

• A note on naming objects: It is okay to use the same name for every arrow. The name **Earrow** is ``recycled'' each time through the loop; it is like a sticker that is moved to a new arrow when that arrow is created.

 •  **Print the value of the electric field at each observation location: print(E)**

**1.4.3** **Arrows representing the electric field**

•  **Run your program. It should calculate and display the electric field as arrows at multiple locations on a circle surrounding the particle. Do not display the relative position vectors -- there are too many of them, and they will clutter up the display.**

• Your program should print the value of each observation location, and Enet at that location.

•  **Look at your display. Does it make sense? Do the magnitudes and directions of the orange arrows make sense?**

 **Make sure everyone in the group agrees that the display looks** **like the field of a dipole!**

**Checkpoint: Ask your instructor to look over your work.**

**1.5** **Adding more locations in a different plane**

• Add code to compute and display the electric field at evenly spaced locations lying in a circle of radius  in a different plane--a plane that also contains the dipole axis. A plane perpendicular to the original plane is a good choice.

• Rotate the ``camera'' with the mouse to get a feeling for the nature of the 3D pattern of electric field.

 **Make sure everyone in the group agrees that the display looks right.**

Discuss your display with a neighboring group.

**Finally, show your display to your instructor.**

**Post a version of your program on your blog and include a screen shot.**

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

**2** **Part II: A Proton Moves Near a Dipole**

Ask your instructor if you are supposed to do this part of the activity.

Create a perpetual loop ( **while True:** ) that continually calculates the net electric field. Place a proton initially at rest at a location on the perpendicular axis of the dipole, then release it, and animate its motion by calculating the electric force, then applying the Momentum Principle  to update the momentum, then updating the position, in a loop. Note that in order to calculate the force on the proton you always have to calculate the electric field of the dipole at the current location of the proton. Leave a trail, so you can study the trajectory:

 proton = sphere(... make\_trail=True) # include a trail

 while True:

 obslocation = proton.pos

 # calculate the electric field

 ...

 # calculate the electric force and update the proton momentum

 ...

 # update the proton's position:

 ...

Try a  of around 1e-17 s, to start with. The trajectory is quite surprising. Think about whether this trajectory is consistent with the Energy Principle.

 If you are using a version of VPython older than 5.70 (execute print(version) to see what version you have installed), you need to create a curve object to represent the trail, and continually append points to the curve object:

 trail = curve(color=proton.color) # this goes before the loop

 while True:

 ...

 trail.append(pos=proton.pos) # in the loop, after updating proton.pos

**2.1** **Playing around**

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

• Add a circle of observation locations in the xz plane.

• If there are red-cyan stereo glasses available (red lens goes on left eye), you can display true stereo by adding the following statement near the start of your program (for more about this, see the section ``Controlling Windows'' in the Visual reference manual available from the Help menu):

 scene.stereo = `redcyan'

• In the Visual reference manual available on the Help menu, in the section on ``Mouse Interactions'', there is documentation on how to make your program interact with the mouse. To wait for a mouse click and get its location, do this in a perpetual loop ( **while True:** ) that calculates the net electric field:

  **while True:**

 # replace obslocation calculation with this:

 obslocation = scene.mouse.getclick().pos # wait for mouse click

 ... # calculate the electric field at obslocation

If instead you want to drag the mouse and continuously update the electric field, remove the arrow and print statements from inside a perpetual loop that calculates the net electric field, and create just one arrow before starting the loop, then continuously update that arrow's attributes:

 Earrow = arrow(axis=(0,0,0), color=color.orange)

 scene.autoscale = False # prevent large mouse moves from autoscaling the scene

 while True:

 rate(30) # check for a mouse move 30 times per second

 obslocation = scene.mouse.pos()

 # calculate Enet

 ...

 Earrow.pos = obslocation

 Earrow.axis = scalefactor\*Enet

• Make a stack of dipoles, and display the electric field.

• Make a ``quadrupole'' out of two opposed dipoles, and display the electric field. The dipoles can be aligned on one axis (+- - +) or they can be placed one above the other, with opposite orientations:

 + -

- +

You'll need to increase **scalefactor** , because the two dipoles nearly cancel each other's fields.

**Electric Field of a Uniformly Charged Rod**

Part I: One Observation Location

**1** **Objectives**

In your textbook you can find the derivation of an equation for the magnitude of the electric field of a uniformly charged thin rod, at observation locations in a plane perpendicular to the rod and bisecting the rod (the ``midplane''). However, what if we need to know the electric field at a location not in the midplane--for example, a location near the end of the rod? In this case, as in many other real-world cases, we can do the calculation numerically, by approximating the rod as a linear array of point charges, and adding up their contributions to the field at the observation location.

In this activity you will:

• Construct a program that does a numerical calculation of the electric field of a uniformly charged rod, in a form general enough that only a one-line change is required to specify a different observation location, or to change the number of point charges in your model.

• Make sure your program gives correct results, by comparing your program's calculation of the electric field in the midplane of the rod to the results of applying the equation derived in the textbook.

• Calculate and display the electric field at locations not in the midplane, where the analytical (symbolic) equation derived in the textbook does not apply.

• Explore the effect of increasing or decreasing the number of point charges used in the model.

• Explore how far from the midplane an observation location must be before the field differs markedly in magnitude or direction from the field in the midplane.

You should finish Part I in 60 minutes or less. Check with your instructor as to whether you are supposed to do Part II or Part III, too.

**2** **Explain and predict program visualization**

A minimal working program is shown below.

**DO NOT RUN THE PROGRAM YET.**  **Read through the program together. Make sure everyone in the group understands the function of *each* program statement. Reading and explaining program code is an important part of learning to create and modify computational models.**

**After reading *every* line of the program, answer the following questions:**

 • SYSTEM: What is the physical system being modeled? Approximately, what should the pattern of electric field look like?  **On the left side of the whiteboard, draw a sketch showing the source charges and how you think the net electric field should look at the specfied observation location.**

 • PROGRAM: How many loops are in the program? What is each loop supposed to do?

 • PROGRAM: Initially, how many point charges are used in the model of the rod?

 • PROGRAM: Initially, how much electric charge does each of the point charges have?

 • PROGRAM: What is the significance of the variable dxq?

 • PROGRAM: Why does the loop start at -L/2 + dxq/2, rather than at -L/2?

 • PROGRAM: Will the program *as it is now written* accurately calculate the electric field of the real system? **DO NOT RUN THE PROGRAM YET.** Study the program again.  **On the right side of the whiteboard, draw a sketch of how the electric field displayed by the program will look, based on your interpretation of the code.**

**Navigate to** [**http://profmason.com/?page\_id=1824**](http://profmason.com/?page_id=1824) **copy and paste the code into an empty vpython window and save it. Remember to give it the extension ``.py''.**

**Run your program *after* everyone agrees on both predictions. Rotate the display to help understand where objects are located. Discuss how closely your prediction of what the program would do matches what you see in the visual output.**

 **Checkpoint: Discuss your answers to the questions above with your instructor.**

**2.1** **Modify the program**

**Complete or correct the program. It may be helpful to look at your program that calculated the electric field of a single point charge, or a dipole.**

**2.1.1** **Notes on the minimal working program**

In the calculation loop, the location of each of the point source charges is calculated as a vector

sourcelocation = vector(xq,0,0)

so that you can use the source location in vector calculations of the electric field  **dE** contributed by that source charge. (Note that since sourcelocation is a vector and not a graphical object, it does not have a position. Just say sourcelocation; don't use .pos .)

In the constants section are the values of the physical constants you will need. You will eventually need a scale factor for representing electric field vectors with arrows, and this has temporarily been given the value of 1.0 (you will need to change this later). You may wish to review the video ``VPython Instructional Videos: 5. Scalefactors'' at http://www.youtube.com/VPythonVideos to see how and why to scale arrows in your program to represent physical vector quantities.

**3** **Numerical experiments**

1.  **Look at your display. Does it make sense? Do the magnitudes and directions of the orange arrows make sense?**

2.  **Rotate the ``camera'' with the mouse to get a feeling for the nature of the pattern of electric field.**

3.  **Change the sign of the charge to negative and run. Does the display make sense? Reset the charge to positive.**

4.  **To check your program, set the  coordinate of the observation location to zero and record the numerically computed electric field. Using your calculator, calculate the electric field at this location using the equation for field in the midplane derived in the textbook:**



* **Does the value calculated by your program have the same order of magnitude as the value you obtain using the analytical equation? If not, check both the program and your calculation.**
* **Vary the number of point charges used to model the rod. What is the minimum number of point charges used to model the rod that gives a result within 1% of the analytical value?**

5.  **With a sufficiently large number of charges in the model to give accurate results, vary the  coordinate of the observation location and note the value of the electric field. About how far from the center of the rod can you get before the magnitude or direction of the field differs significantly from the field at the midplane? (Think of the answer in terms of a fraction of the total length of the rod.)**

**Checkpoint: Ask your instructor to look over your work,** **including your answers to the questions above.**

**4** **Turn In Your Program**

Once you are certain that your program is correct, post it on your blog with an accompanying screen shot and answers to relevent questions.

**5** **Optional: Playing around**

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

• If there are red-cyan stereo glasses available (red lens goes on left eye), you can display true stereo by adding the following statement near the start of your program (for more about this, see the section ``Controlling Windows'' in the Visual reference manual available from the Help menu):

  **scene.stereo = `redcyan'**

• In the Visual reference manual available on the Help menu, in the section on ``Mouse Interactions'', there is documentation on how to make your program interact with the mouse.

To wait for a mouse click and get its location, do this in a perpetual loop (  **while True:** ) that calculates the net electric field:

  **while True:**

 # replace obslocation calculation with this:

 obslocation = scene.mouse.getclick().pos # wait for mouse click

 ... # calculate the electric field at obslocation

If instead you want to drag the mouse and continuously update the electric field, remove the arrow and print statements from inside a perpetual loop that calculates the net electric field, and create just one arrow before starting the loop, then continuously update that arrow's attributes:

  **Earrow = arrow(axis=(0,0,0), color=color.orange)**

 scene.autoscale = False # prevent large mouse moves from autoscaling the scene

 while True:

 rate(30) # check for a mouse move 30 times per second

 obslocation = scene.mouse.pos()

 # calculate Enet

 ...

 Earrow.pos = obslocation

 Earrow.axis = scalefactor\*Enet

**Electric Field of a Uniformly Charged Rod**

Part II: Multiple Observation Locations

**1** **Previous Work**

This activity builds on the activity entitled *Electric Field of a Uniformly Charged Rod* Part I: One Observation Location. Complete Part I before starting this activity.

**2** **Nested Loops**

To see the pattern of electric field near the rod, you need many observation locations. The contribution of all point charges must be calculated and summed up for each observation location. Hence, you need two loops, one inside the other. The outer loop specifies an observation location, and the inner loop calculates the field at that observation location.

To achieve this:

•  **Comment out the statement**

  **Earrow = arrow(pos=vector(-0.3,0.05,0), axis=(0.1,0,0), color=color.orange)**

 •  **Create a loop that varies the observation location, creating a new arrow at each observation location.**

•  **Take all of the statements from the CALCULATIONS section of your program for part I, and indent them inside the loop that varies the observation location.**

If your rod is oriented along the  axis, your code would look something like the following (if your rod is oriented along the  axis or the  axis, you need to make minor modifications to this scheme):

  **Nobs = 12**

 **dxobs = L/Nobs**

 **xobs = -L/2 + dxobs/2**

 **while xobs < L/2:**

  **obslocation = vector(xobs,y,0) # you should choose a value for y**

  **Earrow = arrow(pos=obslocation, axis=(0.1,0,0), color=color.orange)**

  **# your entire existing field calculation loop goes here, indented a level, like this:**

  **Enet = vector(0,0,0) ## starts out at zero and should build up**

  **xq = -L/2 + dxq/2**

  **while xq < L/2:**

  **## Calculate the source location as a vector:**

  **sourcelocation = vector(xq,0,0)**

  **# and so on .....**

  **# increment the observation location:**

  **xobs = xobs + dxobs**

You can easily indent a block of statements by selecting them, then choosing the menu option Format  Indent Region.

**3** An Electron Moves Near a Charged Rod

To apply the Momentum Principle to predict the motion of an electron near a charged rod, you will use the basic iterative calculation structure we have developed so far. At each step, to calculate the new electric force on the electron, you will need to recalculate the electric field of the rod at the electron's current location.

**4** **Motion of an Electron**

**Create a perpetual loop using while True:**

**Assign an initial position and momentum to an electron, and animate its motion by calculating the electric field at the location of the electron, calculating the electric force, then applying the Momentum Principle  to update the momentum, then updating the position (), in a loop, where  can be approximated by** . Note that in order to calculate the force on the electron you always have to calculate the electric field of the rod at the current location of the electron. Leave a trail, so you can study the trajectory:

  **# in the constants section, add qe = -1.6e-19 and me = 9e-31**

 electron = sphere(... make\_trail=True) # specify initial position near the rod

 p = me\*vector(...) # p = mv; specify initial velocity as a vector

 deltat = ? # see discussion below for specifying the integration step size

 while True:

 # calculate the electric field with observation location = electron.pos

 ...

 # calculate the electric force and update the electron momentum

 ...

 # update the electron's position

For a start, set the initial position of the electron to be on the midplane of the rod, with an initial velocity that should give an approximately circular orbit around the rod. Set  **deltat** to be a fraction of the time required to make one orbit. You may want to slow down the plotting by inserting for example **rate(20)** inside the outer loop (in this case, there would be 20 updates per second).

Once you have succeeded in making the electron orbit the rod in the midplane, move the initial position toward one end of the rod and see what kind of orbit you get.

 If you are using a version of VPython older than 5.70 (execute print(version) to see what version you have installed), you need to create a curve object to represent the trail, and continually append points to the curve object:

  **trail = curve(color=electron.color) # this goes before the loop**

 while True:

 ...

 trail.append(pos=electron.pos) # in the loop, after updating electron.pos

**5** **Optional: Playing around**

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

• If there are red-cyan stereo glasses available (red lens goes on left eye), you can display true stereo by adding the following statement near the start of your program (for more about this, see the section ``Controlling Windows'' in the Visual reference manual available from the Help menu):

**scene.stereo = `redcyan'**

• In the Visual reference manual available on the Help menu, in the section on ``Mouse Interactions'', there is documentation on how to make your program interact with the mouse.

To wait for a mouse click and get its location, do this in a perpetual loop (  **while True: )** that calculates the net electric field:

 while True:

 # replace obslocation calculation with this:

 obslocation = scene.mouse.getclick().pos # wait for mouse click

If instead you want to drag the mouse and continuously update the electric field, remove the arrow and print statements from inside a perpetual loop that calculates the net electric field, and create just one arrow before starting the loop, then continuously update that arrow's attributes:

  **Earrow = arrow(axis=(0,0,0), color=color.orange)**

 scene.autoscale = False # prevent large mouse moves from autoscaling the scene

 while True:

 rate(30) # check for a mouse move 30 times per second

 obslocation = scene.mouse.pos()

 # calculate Enet

 ...

 Earrow.pos = obslocation

 Earrow.axis = scalefactor\*Enet

**VPython: Motion of a charged particle in a magnetic ﬁeld**

1 Purpose

You will write a VPython program to simulate the motion of a charged particle immersed in a magnetic ﬁeld. Start with the program shell that is provided at <http://profmason.com/?page_id=1824> (also shown at end of this handout). The program shell draws a ”ﬂoor” and displays a uniform magnetic ﬁeld, which is initially set to < 0 , 0 . 2 , 0 > T.

2 Display a Moving Proton

In this section, you will generate the code to display a proton (shown as a sphere) moving in a straight line. You will not add the eﬀects of the magnetic ﬁeld until section 3.

1. Create a sphere named particle of appropriate radius representing a proton at location <0 , 0 . 15 , 0.3 > , and specify make\_trail=True so that a trail will be left behind the moving particle. Give it an initial velocity of 2 × 106 m/s ( v << c ) in the -x direction, by deﬁning an appropriate vector quantity:

velocity = vector(## you fill this in)

If you are using a version of VPython older than 5.70 (execute print(version) to see what version you have installed), you need to create a curve object to represent the trail, and continually append points to the curve object:

trail = curve(color=particle.color) # this goes before the loop

while True:

...

trail.append(pos=particle.pos) # in the loop, after updating proton.pos

2. Write a loop to move the proton in the direction of its velocity, by following the instructions below.

(If this is a very familiar task to you, you can do it now, without reading the remainder of this step). A computer animation of the motion of an object uses the same principle as a movie or a ﬂip book does. You display the object at successive locations, each at a slightly later time; to your eye, the motion can look continuous. In a program, you calculate the position of the object, and it is displayed in that position. You calculate where the object will be a very short time deltat later, and change its position to the new location, so it is displayed in the new location. You will use the relation between velocity (a vector) and position (a vector) to calculate the position of a moving particle at successive times, separated by a short time step Δ t . Remember that, in vector terms:

In VPython, this translates to a statement like:

particle.pos = particle.pos + velocity\*deltat

where velocity is the vector quantity which you have initialized earlier in the program.

This statement may look odd to you if you have not had much programming experience. The equals sign here has the meaning of ”assignment” rather than equality. This instruction tells VPython to read up the old position of the particle, add to it the quantity, and store the new position as the current position of the particle.This will automatically move the particle to the new position.

3. Read the program shell: What is the value of deltat that is set in the program shell?

4. Inside the loop, update the position of the proton by using its velocity:

particle.pos = particle.pos + velocity\*deltat

5. For this section, you are only generating the code to display a particle moving with constant velocity. In section 6 you will add the eﬀects of the magnetic ﬁeld, so the particle’s velocity may no longer be constant. So far you have updated the particle’s position; think about how you might update the particle’s velocity. DO NOT update the velocity until section 3. (HINT: consider the momentum principle, Δ p = FnetΔ t , where p ≈ mv . Think about how you could adapt the code used to update position to update momentum.) Recall that Δp = FnetΔ t can be written as:

6. Run your program. The proton should move in a straight line. Make sure that it leaves a trail.

7. Check your work so far with a neighboring group, then both groups check with instructor.

3 Add the Force due to the Magnetic Field

Now you will add the force of the magnetic ﬁeld on the moving, charged particle.

1. Before the start of the loop, initialize the particle’s charge, mass, and momentum, where p ≈ mv (since v << c ).

2. Inside the loop, before the line updating the proton’s position, you need to update the particle’s velocity.

3. How did you decide to update the particle’s velocity in section 2

• What will you need to calculate in order to update the particle’s velocity?

• You may decide to ﬁnd the (vector) force of the magnetic ﬁeld on the moving, charged particle.

 A cross product A × B can be calculated in VPython as cross(A,B), if A and B are vector quantities.

• Compare you plan with another group, and then both groups check with an instructor.

4. Complete the loop in the program shell with the plan that you checked above.

4 Experiments

Do the following experiments and answer the questions. If the program runs too fast, decrease deltat.

1. What is the approximate radius of the path?

2. What happens to the radius of the path if you double the initial speed? Halve the initial speed?

3. Reset the speed to 2 × 106 m/s. Change the while statement to while t < 3 . 34 e − 7: and run the program. The proton should make just one complete orbit. If not, review your calculations. If you double the initial speed, the radius of the path will also double: How far around will the proton get in the time 3 . 34 × 10−7 seconds? Try it. What do you ﬁnd? This striking behavior is the basis for the cyclotron (see textbook).

4. Change the proton’s initial velocity to be in the same direction as the magnetic ﬁeld. What kind of path does the proton follow? Why?

5. Restore the velocity to its original value, and change the while statement to do 5 complete circles.

Add a signiﬁcant +y component to the proton’s velocity. Now what kind of a path does the proton follow? Why? What is the connection with question 3

6. Change the proton to an antiproton, a particle with the same mass as the proton, but a charge of −e . What changes?

7. Before you post your program, check that: you have changed the proton to an antiproton, restored the original x-component of the velocity and added a signiﬁcant y-component to the velocity, and that the while statement is set for 5 circles.

5 Report

Submit answers to the questions above, program code and an animation of your program to your blog.