Lab 1: Kinematics, Intro to Computer Programming

Instructions

1 Kinematics

- You will spend the first ~hour of lab working on a tutorial from the UW “Tutorials in Introductory Physics” workbook called “Representations of Motion”.

- The goal of this part of the lab is to get more comfortable with the relation between position, velocity, and acceleration. You will make predictions for different types of motion and then test your predictions.

- **Materials and Equipment:** motion detector (provided), tutorials workbook (bring it with you to lab), some scratch paper and a pen/pencil.

- Choose a computer workstation with your group (2-3 students) and work through the tutorial (pages 7-10 in workbook).

2 Introduction to Programming with VPython

This exercise is based on the tutorial developed by the Matter and Interactions Group at NCSU. It will guide you through some basics of programming in VPython. VPython is a programming language that allows you to very easily make 3D graphics and animations, so we’ll be using it throughout the course to analyze and visualize the behavior of physical systems.

For this part of the lab, you can work in the same group with the same computer as in part (1). At the end of the lab period, your TA will show you where to download VPython from the web (www.vpython.org). It’s relatively easy to install on your own computer.

2.1 Your First Program

- Double click on the “IDLE for Python” icon in the “Start” menu. This starts the editing environment (IDLE) for VPython.

- Enter the following line of code in the IDLE editor window:

```python
from visual import *
```

Every Vpython program begins with this line of code. This line tells the program to use the 3D module called “visual” (which is what the “V” stands for in VPython!)

- Save your program: from the “File” menu, select “Save”. Save your file in the “p13_Lab1” directory including one of your last names in the filename. For example, I would name my file “millan_vectors.py”. **Note: You MUST include the .py extension or VPython will not recognize your program!**

2.2 Creating 3D Objects

- VPython has built-in commands to generate simple 3D objects. On the next line of your program, type:

```python
sphere()
```
Now Run the program by pressing F5 on the keyboard. Two new windows should pop up. One is the 3-D graphics window which should display a sphere. The other is the Python Shell window which displays error messages and printed text (if you tell the program to print something, for example).

You can use the mouse to zoom in or rotate the view in the 3D graphics window.

- Hold down both mouse buttons and move the mouse up or down to zoom in or out.
- Hold down the right mouse button and move the mouse to rotate the camera view about the center of the scene.

To kill the program, close the graphics window. Do not close the Python Shell window since it provides useful information such as error messages.

STOP!
Checkpoint 1: Run your program for your TA so he/she can make sure you made it this far and don’t have any questions before proceeding.

2.3 Error Messages

- As mentioned before, the Python Shell window will display error messages.

- To see an example of an error message, let’s make a spelling mistake in the code. Change the first line of your program to the following:

  from bisual import *

- Run the program. You should see a message in red text in the Python Shell window. The message gives the filename, line number of the error and a description of the error (in this case, “ImportError: No module named bisual”).

- Note: you may not understand the error messages, but it is important that you know there is an error. You can usually find the error and fix it pretty quickly with some practice. One hint that will make your life easier: make only a few changes to your program at a time and run it to check for errors. If there is an error, it will be easier to track down!

- Correct the error in the first line of your code. Run the program to make sure the error has been fixed.

2.4 Object Attributes

- We can change the position, radius, or color of our sphere by giving it attributes. Change the last line of your program to read:

  sphere(pos=vector(-5,2,-3), radius=0.40, color=color.red)

- Run the program

  This statement gives the sphere three attributes
- **pos**: The position vector of the center of the sphere relative to the origin at the center of the screen.
- **radius**: the sphere’s radius
- **color**: the color of the sphere. Other colors are blue, green, cyan, magenta, yellow, orange, black, or white.

- Experiment with different values for the attributes of your sphere.
- Change the attributes of your sphere to:
  \[\text{sphere}(\text{pos}=\text{vector}(2,4,0), \text{radius}=0.20, \text{color}=\text{color.white})\]
- Create a second sphere with position (-3,-1,0) and radius=0.15. Make this sphere green. You should now see two spheres when you run your program. In later exercises, the white sphere will represent a baseball and the green sphere will represent a tennis ball.

### 2.5 Arrows

Another object we can create in our 3D graphics window is an arrow. We will use arrows to depict vector quantities in later programs.

1. Creating an arrow
   - Add the following line to your program and run the program:
     \[\text{arrow}(\text{pos}=\text{vector}(2,-3,0), \text{axis}=\text{vector}(3,4,0), \text{color}=\text{color.cyan})\]
   - To illustrate the difference between the **pos** and **axis** attributes, add another arrow to your program:
     \[\text{arrow}(\text{pos}=\text{vector}(3,2,0), \text{axis}=\text{vector}(3,4,0), \text{color}=\text{color.red})\]
   - Compare the two arrows. What is different? What is the same? Experiment with the **pos** and **axis** attributes and answer the following questions:
     (a) What does the **pos** attribute of an arrow describe?

     (b) Define the **axis** attribute of an arrow?

     (c) What position would you give a sphere so that it would appear at the **tip** of the red arrow? Check your answer with your program!

2. Scaling an arrow’s axis
   - Since the axis of an arrow is a vector, we can perform scalar multiplication on it. Change the last line of your program to the following:
     \[\text{arrow}(\text{pos}=\text{vector}(3,2,0), \text{axis}=-0.5*\text{vector}(3,4,0), \text{color}=\text{color.red})\]
   - Notice that the red arrow now points in the opposite direction because of the minus sign. It is also half as long as the cyan arrow. Multiplying by a scalar changes the length of an arrow.
• For the next section, you’ll only need one arrow. Instead of deleting one of the arrows from your program, let’s “comment out” a line. Change the second to the last line of your program to the following:

\[
\# \text{arrow} \left( \text{pos}=\text{vector}(2,-3,0), \text{axis}=\text{vector}(3,4,0), \text{color}=\text{color.cyan} \right)
\]

The pound sign at the beginning of the line causes the program to ignore this line. Run your program to verify that the cyan arrow is no longer there.

3. Position vectors

• We can use arrows to represent the position vector of an object or the relative position between two objects. Recall that a relative position vector that starts at a position \( \vec{A} \) and ends at a position \( \vec{B} \) can be found by “final minus initial”, or \( \vec{B} - \vec{A} \).

• Make an arrow that represents the relative position of the tennis ball with respect to the baseball. That is, make an arrow whose tail is at the position of the baseball (white sphere) and tip is at the position of the tennis ball (green sphere).

• You will have to determine the \( \text{pos} \) and \( \text{axis} \) attributes of the arrow. Remember, \( \text{pos} \) gives the position of the arrow tail, and \( \text{axis} \) describes its length.

2.6 Naming Objects and Using Object Attributes

• We can give names to the objects in a program. This makes it easier to refer to them or their attributes later in the program. Change the “sphere” lines of your program to the following:

\[
\text{baseball} = \text{sphere} \left( \text{pos}=\text{vector}(2,4,0), \text{radius}=0.20, \text{color}=\text{color.white} \right)
\]
\[
\text{tennisball} = \text{sphere} \left( \text{pos}=\text{vector}(-3,-1,0), \text{radius}=0.15, \text{color}=\text{color.green} \right)
\]

• We can now refer to the attributes of each sphere by writing, for example, “tennisball.pos” to refer to the tennis ball’s position. That is, “tennisball.pos” is a vector with value \((-3,-1,0)\). Similarly, “tennisball.radius” is a number with value 0.15. You can check this by printing these attributes. Add the following line to your program:

\[
\text{print tennisball.pos}
\]

• Run the program. You should see the printed vector in the Python Shell window.

• Give a name to your arrow also.

• Since we can refer to the attributes of objects symbolically, we want to write symbolic expressions for the “axis” and “pos” of the arrow. The expressions should use general attribute names in symbolic form, like “tennisball.pos” or “baseball.pos”, not specific numerical values. This way, if the position of the tennis ball or baseball is changed, the arrow will still point from the baseball to the tennis ball.

  – Change the “pos” attribute of your arrow to a symbolic expression (not numbers) so that it still points from the baseball to the tennis ball.

  – Run your program. Examine the 3D graphics window carefully to make sure the red arrow still points from the baseball to the tennis ball.
Change the position of the baseball to (-4,-2,5) and the position of the tennis ball to (3,1,-2). Note that you are giving both z-coordinates also! Run your program and make sure the arrow still points from baseball to tennis ball.

STOP!
Checkpoint 2: Ask your instructor to check your program for credit.

2.7 “Ball in a Box” Assignment

In this exercise, you will create an animation of a ball bouncing in a box. You can finish this at home if you run out of time in your lab section. The program is due at the beginning of your next lab period (show your TA your animation to get checked off). Your group may work together outside of class and turn in the assignment for group credit.

1. Create a new program and make a sphere object:
   \[
   \text{ball = sphere(pos=vector(-5,0,0), radius=0.5, color=color.red)}
   \]

2. Create a “wall” by making a box object:
   \[
   \text{wallR = box(pos=vector(6,0,0), size=(0.2,4,4), color=color.green)}
   \]

3. Animating the ball
   - We would like to make the red ball move across the screen and bounce off the green wall. We can think of this as displaying “snapshots” of the position of the ball at successive times as it moves across the screen. To specify how fast the ball moves, we need to specify its velocity and how much time has elapsed.
   - First, we’ll define a short time interval. We’ll call it “dt”. Type this line at the end of your program:
     \[
     \text{dt = 0.05}
     \]
   - Now define the ball’s velocity. Add the following line:
     \[
     \text{ball.velocity = vector(2,0,0)}
     \]
     Notice that we have defined the velocity as a vector. In this case, we’ve specified that the ball moves only in the x-direction (the “y” and “z” components of the velocity are zero).
   - Run your program. Nothing happens! That’s because we haven’t yet given the computer instructions on how to use the velocity to update the ball’s position.
   - Since distance=speed*time, we can calculate how far the ball moves (its displacement) in the time interval, dt. To find the ball’s new position, we add the displacement to the old position:
     \[
     \text{ball.pos = ball.pos + ball.velocity*dt}
     \]
     (Note that, since ball.velocity is a vector, ball.position is also a vector.)
   - The last line you added to your code may look a bit odd to you. How can we set \text{ball.pos} to \text{ball.pos} plus something else? Isn’t that like saying, “1=1+2” which is clearly false? NO! The meaning of the equal sign here is different than its meaning in arithmetic. In the program, the equal sign means, “assign a value to this variable”. You are really giving the following instructions: “Find the location in memory
where the value of the variable *ball.pos* is stored. Read up that value, add the value *ball.velocity*dt* to it, and re-store the new value in memory.” In other words, we are telling the computer to update the value of *ball.pos* to a new value. This is how we animate the ball. We give the computer successive updates to the ball’s position which is changing in time as the ball moves.

- Run your program. Again not much happens! The problem is that the program only took one time step. We told it to update the ball’s position only once. But, we want the program to take many timesteps so the ball keeps moving. To accomplish this, we use a *while loop*.

- Delete the last line of your program and type:
  ```python
  while (1=1):
  ```
  Indented under this “while” statement, type:
  ```python
  ball.pos = ball.pos + ball.velocity*dt
  ```
  Note: the IDLE editor should automatically indent the cursor for you when you press enter after the “while” statement. Alternatively, you can press the TAB key. The indentation is important. It tells the computer which statements are “inside” the loop. All indented statements after the “while” statement will be executed every time the loop is executed.

- Your program should now look like this:
  ```python
  from visual import *
  ball = sphere(pos=vector(-5,0,0), radius=0.5, color=color.red)
  wallR = box(pos=vector(6,0,0), size=(0.2, 4,4), color=color.green)
  dt=0.05
  ball.velocity = vector(2,0,0)
  while (1=1):
      ball.pos = ball.pos + ball.velocity*dt
  ```

- Run your program. You may see the ball moving very rapidly across the screen. On my computer, it’s too fast to see. To slow it down a little, insert the following statement inside the loop (after the “while” statement):
  ```python
  rate(100)
  ```
  You should now see the ball move from left to right across the screen!

4. Making the ball bounce: Logical tests

When you ran your program, you probably noticed that the ball didn’t bounce off the wall at all. It just kept going left to right. That’s because we have to tell the computer to make the ball bounce off the wall.

- First, we need to detect a “collision” between the ball and the wall. A simple approach is the compare the x-coordinate of the ball to the x-coordinate of the wall, and reverse the x-component of the ball’s velocity at the moment they are equal. We can use a logical test to do this:

  ```python
  if ball.x > wallR.x:
      ball.velocity.x = -ball.velocity.x
  ```
The indented line after the “if” statement is only executed if the statement is true. If the result of the logical test is false (that is, if the x-coordinate of the ball is not greater than the x-coordinate of the wall), the indented line will be skipped.

- We would like this logical test to be carried out at each time step (every time the ball is at a new position). Therefore, we must insert this statement inside the “while” loop. Your program should now look like this:

```python
from visual import *
ball = sphere(pos=vector(-5,0,0), radius=0.5, color=color.red)
wallR = box(pos=vector(6,0,0), size=(0.2, 4,4), color=color.green)
dt=0.05
ball.velocity = vector(2,0,0)
while (1=1):
    ball.pos = ball.pos + ball.velocity*dt
    if ball.x > wallR.x:
        ball.velocity.x = -ball.velocity.x
```

- Run your program. The ball should now reverse direction when it encounters the wall. Notice that our logical test is not very sophisticated since the ball penetrates the wall part way before reversing direction. We could fix this but won’t for now.

5. Add another wall at the left side of the display and make the ball bounce off that wall also so it bounces between the two walls.

6. Make the ball move at an angle by changing ball.velocity such that it has a non-zero y-component. Now that the ball moves at an angle, it might be missing the wall! Make the walls bigger (change size = (.2, 12, 12) in the “box” statements). We’ll also fix this later by adding a horizontal wall above the other walls.

7. Visualizing velocity and leaving a trail

- Before adding more walls, let’s first add an arrow to show the velocity of the ball. Create an arrow before the “while” statement whose tail is at the ball location to represent the ball’s velocity:

```python
bv = arrow(pos=ball.pos, axis=ball.velocity, color=color.yellow)
```
• Update the position and axis of the arrow by adding the following lines inside the “while” loop:
  
  \[
  \text{bv.pos} = \text{ball.pos} \\
  \text{bv.axis} = \text{ball.velocity}
  \]

• Run your program. The arrow should now move with the ball and should change direction every time the ball hits a wall.

• To visualize the trajectory of the ball, we can make it leave a trail. To do this we create a “curve” object (before the “while” loop):
  
  \[
  \text{ball.trail} = \text{curve(color=ball.color)}
  \]

  At the end of the while loop (inside the loop), add the line:
  
  \[
  \text{ball.trail.append(pos=ball.pos)}
  \]

  This will add a point to the trail at the latest ball position.

8. Add top and bottom walls to your program so that they touch the horizontal walls. Give these a different color to make the 3D nature of the object more visible. Make the ball bounce off to top and bottom walls by adding the appropriate “if” statements in the “while” loop.

9. Add a back wall and an “invisible” front wall. That is, for the front wall, don’t create an actual wall object, but include an “if” statement to prevent the ball from coming through the front.

10. Give your ball components of velocity in the z-direction as well to make it bounce off all six walls.

Important!! Don’t forget to finish the “ball in a box” assignment before your next lab session. You will have to copy or email your program file to transfer it to your own computer. Run your program for your TA to get full credit. If you need help finishing it or installing VPython during the week, you can go to any of the Instructor office hours! This assignment counts for 40% of your Lab 1 grade!