

Physics 231 – Lab 3

Predicting Motion for Non-Constant Forces

There are three parts to this lab; they all deal with motion under non-constant forces. First, you'll experimentally measure the motion of a cart on a spring. Next, you'll create a VPython simulation to model that motion. Finally, you'll model the interaction of the Earth and the Sun.

(Equipment: force probe, motion sensor, cart & force-probe bracket, spring, scale)

Objectives

In this lab you will:

- Observe and model the motion of a cart connected to a spring
- Model motion of two objects interacting gravitationally

Background

A typical program will have the following four sections:

- (1) Setup statements
- (2) Definitions of constants
- (3) Create objects and specify initial conditions
- (4) Repeated calculations to predict motion of objects (in a loop)
 - Calculate the (vector) force acting on each object.
 - Update the momentum of each object using $\vec{p}_f = \vec{p}_i + \vec{F}_{\text{net}}\Delta t$
 - Update the position of each object using $\vec{r}_f = \vec{r}_i + \vec{v}_{\text{avg}}\Delta t$, where $\vec{v}_{\text{avg}} \approx \vec{p}_f/m$.

It is helpful to label these sections with comments by using lines start with the pound sign (#).

I. Spring Force on a Cart

A. Experiment

First, you will determine the stiffness of a spring and then use a motion sensor to measure a cart's position and velocity while it is acted on by the spring.

- **Spring Constant and Equilibrium Position**
 - First, just to get it out of the way, determine the mass of the cart (there's a scale at the back of the room) and enter it in WebAssign.
 - Open CartOnSpring.clbs from the Phys 231 folder within Physics Experiments (on the desk top).
 - While nothing is attached to the hook of the force probe, zero it (the Zero button near the big green "Collect" button.)
 - You'll notice that one of the black ends of the cart has a little hole at the top – orient the cart with *that* end facing the force probe / spring, and hook the spring through the hole so the spring connects the cart to the force probe. Now, move the cart so its far edge (not the one connected to the spring) sits at the four different positions listed in WebAssign and record the corresponding spring forces that LoggerPro reports.
 - Next, in WebAssign, plot these positions and forces to determine the equilibrium position (intercept) and the spring's stiffness (slope.)
- **Cart's Motion when experiencing Spring Force**
 - Now, launch the cart toward the motion detector and use loggerPro to record its motion up and then back down the track (so you don't miss any of the motion, wait to launch until you hear the detector start taking data – the cart easily 'jumps the rails', so you may need to give this a few tries.)
 - Zoom in on the region of the position, velocity, and force graphs *just for the time when you're hand wasn't touching the cart* (convince yourself that you can recognize

when that is – for example, the velocity is becoming more positive while you’re pushing, and starts heading back down the moment you stop pushing.) Remember, you can select “examine” under the “analyze” menu to see the coordinates of points on the plot.

- Pick a time slightly *after* the cart left your hand, and determine the cart’s position and velocity values; ditto for a time slightly *before* you caught the cart. You’ll record these values in WebAssign.
- Minimize the LoggerPro screen, *but don’t actually close the program* – you’re going to be comparing these graphs with ones you’ll generate with a simulated cart on a spring.

B. Computer Model

Now, you will see how well you can model the cart’s motion in a computer simulation.

- Make a copy of one of your team members’ program “fancart.py” from last week. Call it “CartSpring.py”. Put your names in a comment line near the top, and if you didn’t do so before, label the sections of the program with comments (Setup, Constants, Objects & Initial Conditions, and Calculations.)
- In the constants section, define the stiffness of the spring (k_s) and give it your measured value. Also, define the vector position of the cart when the spring is not stretched (`unstretched`) and give it the x_o value you’d measured. You have to fill in the x coordinate.

```
unstretched = vector( , 0, 0)
```

Make the following changes in the section where the objects are created and given their initial conditions to match those measured experimentally:

- Shift down the y coordinates of the cart and track so the cart is centered on zero and the track just touches its bottom.
- Shift over the x coordinate of the track so that its left end at $x = 0$.
- Set the x coordinate of the cart to the initial value measured in the experiment, x_i .
- Use the experimentally measured values of the mass, m , and velocity’s x -component, v_{xi} .
- Set the initial value of the time to the one from the experiment, t_i .

Make the following changes to the section with the repeated calculations:

- Change the final time for the loop to the one from the experiment, t_f .
- So that the cart is acted upon by the spring force ($\vec{F}_{net} = -k_s \left(\vec{r} - |\vec{L}_o| \hat{\vec{L}} \right)$ where in our case, and $\vec{L} = \vec{r}_{cart}$ and so $\hat{\vec{L}} = \hat{r}_{cart} = \vec{r}_{cart} / |\vec{r}_{cart}|$), remove the old lines that had calculated the force(s) and insert inside the top of the loop the lines

```
delta_L = mag(cart.pos) - mag(unstretched)
L_hat = cart.pos/mag(cart.pos)
Fnet = -ks * delta_L*L_hat
```

Pause and think: If it stretches the spring beyond its equilibrium length, what direction does the force point?

- Run the program to see if your simulated cart has the right qualitative behavior – the motion looks like that you observed experimentally.

To make a slightly stronger comparison between experiment and simulation, you'll have VPython generate graphs of position and velocity vs. time just as LoggerPro had.

- In the setup section, add the following line to load the graphing functions:

```
from visual.graph import *
```

- In the section where objects are created, add the following lines to create curves for x vs. t and v_x vs. t graphs:

```
xvst = gcurve(color=color.red) # initializes x vs t graph  
vbst = gcurve(color=color.green) # initializes v vs t graph
```

- Inside the beginning of the loop, add the following lines to add points to these curves on the graphs:

```
xvst.plot(pos=(t, cart.x)) # adds new position to x vs t graph  
vbst.plot(pos=(t, cart.p.x/cart.m)) # adds new velocity to v vs t graph
```

In these two commands, you're specifying the points to plot on the graphs – horizontal coordinate (time) and vertical coordinate (x or v_x).

C. Model – Experiment Comparison

Bring back up LoggerPro so you can compare the graphs of your experimental data and your computationally simulated data. When you run it, does the computer model qualitatively reproduce the graph from experiment? Remember, it should only correspond with the segment from t_i to t_f .

- To make your comparison a little more quantitative, insert a line just outside the bottom of the loop that prints the x components of the cart's final position and velocity.
- If these values that the simulation determines differ by much more than 10% from the ones you'd measured experimentally, find and fix your error. (Common errors: taking t_i or t_f to be times when a hand *was* touching the cart, or not giving the simulated cart the real cart's mass, initial position, or initial velocity.) Enter these values into WebAssign.

Save the completed “CartSpring.py” and upload it in WebAssign.

II. Planetary Orbit Simulation

In this part, you will model the motion of a planet around a star (initially, the Earth around the Sun).

A. Useful Data

$$G = 6.7 \times 10^{-11} \text{ N} \cdot \text{m}^2 / \text{kg}^2$$

$$M_{\text{Earth}} = 6 \times 10^{24} \text{ kg} \quad R_{\text{Earth}} = 6.4 \times 10^6 \text{ m}$$

$$M_{\text{Sun}} = 2 \times 10^{30} \text{ kg} \quad R_{\text{Sun}} = 7 \times 10^8 \text{ m}$$

$$\text{Distance from Earth to Sun} = 1.5 \times 10^{11} \text{ m}$$

$$\text{Orbital Period} = 1 \text{ yr}$$

B. Planning

Before writing a program, you need to figure out two numbers that you will need.

- The Earth takes one year to go around the Sun in a nearly circular orbit. Making the approximation that the Earth's speed is constant, determine the initial speed, $|\vec{v}_{\text{initial}}|$, you should give the Earth in a computer model. You may wish to work it out on a whiteboard.
- Now to set a time step. On the one hand, you don't want it so large that the Earth moves enough that the force on it changes appreciably during one step; then again you don't want it so small that you're wasting time waiting for the computer to finish its billionth calculation. Now, the Earth goes around the sun in 365 days, so in a time step of one day, it changes its direction by about 1° (it's no coincidence that there are 365 days a year and 360° to a circle) or on order of a few percent; that's almost small enough to be our time step. How about dividing the orbit / year into 1000 steps. Determine how many seconds Δt would be then.

C. Circular Orbit

- Open a new file with Idle for Python and save it as "EarthSun.py"; alternatively, open your old solar-system model from Lab 1, resave it as "EarthSun.py", and then modify it appropriately.
- Use comments to name the program, insert your names and label the sections of the program as you write the program.
- Enter the setup commands.
- Define the gravitational constant, G (in standard units.)
- Create spheres called `Sun` and `Earth` with different colors. So that you can see them, make the radius of the Sun 20 times its actual value and the radius of the Earth 1000 times its actual value. Place the Sun at the origin and the Earth initially at its actual distance, straight out along the x axis. Set the masses of the objects (`Sun.m` and `Earth.m`).
- Set the initial momentum of the Earth to be purely in the y direction and incorporate the speed calculated on the previous page. Set the initial value of the time (`t`) to zero. Define the time interval (`deltat`) using the value determined on the previous page.

The most complicated part of the program is the repeated calculations.

- Start the loop with the following statement. Choose a final time so that the Earth will go around the Sun at least twice.

```
while t <  :
```

- Inside the loop (indented), calculate the gravitational force on the Earth by the Sun. The power of a number is calculated using two asterisks. For example, "`rmag**2`" is the square of `rmag`.

The function “`sqrt()`” finds the square root of the argument. Here’s what you’ll need to enter in the code, and what it means in mathematical notation:

Code

```
r = Earth.pos - Sun.pos
rmag = sqrt(r.x**2 + r.y**2 + r.z**2)
rhat = r/rmag
F = -G * Earth.m * Sun.m*rhat / rmag**2
```

Math

$$\vec{r} = \vec{r}_{Earth} - \vec{r}_{sun}$$

$$|\vec{r}| = \sqrt{r_x^2 + r_y^2 + r_z^2}$$

$$\hat{r} = \vec{r} / |\vec{r}|$$

$$\vec{F}_{Earth \leftarrow Sun} = -G \frac{m_{Sun} m_{Earth}}{|\vec{r}|^2} \hat{r}$$

- Also inside the loop:
 - (a) Use the momentum principle to update the momentum of the Earth experiencing the gravitational force.
 - (b) Use the “position update relation” to update the Earth’s position.
 - (c) update the time.
 - (d) Add a “rate” command so the simulation runs at a reasonable speed.

Making it Pretty

It is often useful to be able to see the path an object’s taken. To make a trail behind the Earth, do the following:

- In your definition of the Earth object, inside the brackets where you’ve set its size and other attributes, add

```
Make_trail = true
```

As objects move, the display may jitter because VPython adjusts the size of the scene to make sure everything is visible.

- To turn off this behavior, enter the following line just *after* creating all of the objects. It is important to put this statement after the initial creation of the objects so that everything is in the scene.

```
scene.autoscale = 0
```

When you have the simulation running, make sure it’s behaving qualitatively as it should – the orbit should indeed appear to be circular (note: VPython renders images in perspective; looked at from an angle, a circle looks like an ellipse.)

Before going on.

- **Why the small timestep:** You’ve made a point of using a time step, `deltat`, that’s small enough that the Sun’s force and Earth’s momentum hardly change during it. As the text suggests, making too large a time step will get ‘unrealistic behavior.’ Take a moment and see just how too-large a time step ruins a simulation; see what happens when you make the time step, `deltat`, larger and larger. (Of course, once you’re done, return your time step to its original, reasonable size.)
- **Why neglect the Sun’s motion:** Even though the Earth’s pull on the Sun is just as strong as is the Sun’s pull on the Earth, we’ve neglected that in the simulation – you don’t update the *sun*’s position inside the loop. Discuss with your group why that’s fairly realistic anyway - why isn’t the Sun’s position significantly affected by the interaction?

D. Non-Circular Orbits

One of the virtues of a simulation is that you can experiment with it in ways that you could never do with a real system. Now, you'll explore what the other types of orbits are possible.

- Set the initial speed to 1.2 times the Earth's actual speed. Increase the maximum time to a several years.
- Vary the time interval to make sure that it is small enough that the path is unaffected.
- It may be helpful to turn autoscaling back on. You can do this by changing the value of `scene.autoscale` to one.

Question: In WebAssign, indicate the shape of the orbit.

Different speeds – Different shapes. Increasing the earth's speed by 20% had qualitatively changed the shape of the orbit, try increasing the speed more and more and see what other shape you can get. Changing the speed changes the trajectory's "eccentricity" from that of a circle, to an ellipse, a hyperbola, and finally a parabola – the "conic sections" from high school geometry. While you're at it, you might want to see what would happen if the Earth's speed *slowed down* (When you're done experimenting, change the initial speed back.)

E. Force and Momentum Vectors

In physics, we like to visualize not just a system's changing positions, but the relevant properties like momentum and force – that helps us to understand the interplay and effects of these properties. So we make 'cartoons' with arrows representing these vector properties overlaid on our images and movies. So, now you will add vectors representing the momentum of the Earth and the Sun's force on the Earth. It will be convenient to place the tails of both of these vector at the Earth's location.

So far, everything visualized in our simulation is *spatial* with units of meters (how big the sun is, how far the Earth is from it...), but these momentum and force vectors are completely different beasts with completely different units. There's no guarantee that the proper scale for displaying the *locations* of things is also the proper scale for displaying their *momenta* or the *forces* acting on them. So you'll need to use *scale factors* to make the force and momentum arrows reasonable lengths.

View Video 5. Scalefactors <http://www.youtube.com/vpythonvideos>

- *Before* the start of the "while" loop, create the arrows to be displayed with the following lines:

```
forcearr = arrow(color=color.red)
momarr = arrow(color=color.green)
```

- Also, create the "scale factors" `fscale` and `pscale` and set them to zero *before* the "while" loop. The use of these will be explained below.
- Leaving the `pos` and `axis` attributes of the arrows blank will create arrows of zero length at the origin. After updating the Earth's position *inside* the "while" loop, adjust the positions of the force and momentum arrows so that their tails are at the Earth's position:

```
forcearr.pos = Earth.pos
momarr.pos = Earth.pos
```

- You will also need to adjust the `axis` (length) of each arrow to represent the force and momentum vectors using the scale factors:

```
forcearr.axis = fscale * F
momarr.axis = pscale * Earth.p
```

The scale factors `fscale` and `pscale` are necessary in order to make the arrows fit into your scene. Your scene is a few times 10^{11} meters across, so you need to scale your arrows to be about 10^{10} in length.

- Inside the "while" loop, print the force and the momentum to see roughly how big they are.
- Adjust the both `fscale` and `pscale` so that the vectors are reasonable sizes. For example: Suppose the force components are about 10^{31} (Newtons), and you want to scale down to 10^{10} (meters). You would need `fscale` to be about 10^{-21} , so that multiplying it times force gives about 10^{10} .

Answer the following questions in Lab 3 on WebAssign.

- Question:** When the Earth is to the right of the Sun, in what direction is the *force on the Earth*?
- Question:** Again, when the Earth is to the right of the Sun, in what direction is the *momentum* of the Earth?
- Question:** If the orbit isn't already elliptical, make it so by adjusting the Earth's initial speed. How does the momentum depend on the distance from the star?

Save the completed "EarthSun.py" (with elliptical orbit) and upload it via Lab 3 in WebAssign. If you aren't using your own laptop, email yourself copies of CartSpring.py, Binary.py and EarthSun.py.