How to make decent computer models of motion:
The Euler-Cromer method is a variation on the Euler method of numerical integration used in calc classes.
Do the following:

- Define values of constants like g or k (spring constant)
- Specify initial conditions (initial mass, position, velocity, etc)
- Specify a tiny time interval Δt. This time interval is supposed to be small enough such that acceleration, force, and velocity are approximately constant for that tiny amount of time. In practice you guess initially and refine your guess based on how the program runs.
- Create an iterative loop that does the following:
    - Calculate net force vector based on *current* position
    - Update the momentum of the system using $\vec{p}_f = \vec{p}_i + \vec{F}_{net}\Delta t$
    - Compute current avg velocity using $\vec{v}_{avg} \approx \frac{\vec{p}_f}{m}$
    - Update the position using $\vec{r}_f = \vec{r}_i + \vec{v}_{avg}\Delta t$

Nothing here will ever be exactly perfect. That said, you can keep slicing up using a smaller Δt to get higher quality *approximate* answers.

Ready to code? Might as well dive in…
Go to http://vpython.org/ and read the page where it says "How to Get Started".

Following the instructions, I went to glowscript.org. I clicked sign in and it let me sign in with google id. I don't particular like having to use a google Id but I figured I wanted this done so I did it. I understand if you are morally opposed. Once I got signed in I first wrote the cheesiest program ever:

```
box()
```

I then ran the code. Once you hit run, try swiveling the point of view by holding down the right mouse button and moving the mouse. Try zooming by holding both mouse buttons and moving the mouse. Try changing the size of the black window by dragging on the bottom right corner.

Once you feel good, go on to the next page. Note: you might also click on the HELP link in the upper right corner of the glowscript page. This should link you to a page with all kinds of documentation on different types of objects. There are also some links to training videos and a tutorial if you scroll down a bit. Sometimes the documentation is not quite perfect but it is free…

From a book called Matter and Interaction more video tutorials can be found here:
www.Vpython.org/video01.html
www.Vpython.org/video02.html
www.Vpython.org/video03.html
www.vpython.org/video04.html
I started watching the first vid and it seemed pretty instructive and easy to follow.

Now try writing the following code. **Pay close attention: capitalization, punctuation, and indentation mistakes can cause the code to fail.** *Sometimes* capitalization or indentation doesn't matter, but for now, copy stuff *exactly* as shown.

```
from visual import *
ball=sphere(pos=vector(1, 2, 3),
            color=color.orange,
            radius=0.5)
velocity=vector(1, 3, 1)
arrow(pos=ball.pos,
        axis=velocity,
        color=color.green)
deltat=0.1
t=0
while t<80:
    rate(10)

    ball.pos=ball.pos+velocity*deltat
    t=t+deltat
```

Run it and see what happens. Rather than cut and paste, actually type in the code line by line. You might make a few typos and finding those typos will ultimately be valuable practice. What out for indentation! When I first typed it up, I had a heck of time before realizing that indentation seems to matter…a lot!

Once your code runs, try playing with the numbers one at a time to see what happens. When you've got this down, go to the next page.

On the following page is an example of some code I wrote up. The example draws two superballs (very bouncy balls). The balls are initially located at different positions. The two balls are attracted to each other by a gravitational force (we will learn more about this later). For now, know the *magnitude* of the gravitational force is given by

$$F = \frac{Gm_1m_2}{r^2}$$

This gravitational force acts directly towards the other ball. For spice, I assumed if the superballs hit they bounce off each other. I assumed a perfectly elastic collision. I chose to make the balls reverse direction with equal speed upon impact. WARNING: this is only true in certain special cases…

This code is not the smartest or most elegant, but it does work. In theory you are already able to log in to glowscript, run the code, and swivel/zoom the simulation. Now try playing around with it by changing some numbers slightly.

- First try playing with the rate (try 5 or 100). Now set it back to 10.
- Try playing with the time step deltat. What happens if you make it 1 or 0.1.
- Try changing the line "while t<200" to "while t<20"

Think: suppose you want to cut the time step in half to increase the quality of the approximation used in the simulation. You still want the simulation to run for the same total time on the screen. Which other number do you need to change (the rate number of the while number)? Try it and see.

- Try changing the ball radii and color.
- Try changing the initial positions of each ball.
- Try changing the initial momentum of each ball (the p1 and p2 vectors in lines 5 & 6).
- Try changing the numbers for interval or retain.
- Try cutting out all text in the lines with trail_type, interval, and retain for one of the rocks.
- Try changing the initial value of the vector r to something other than (0, 0, 0). When you run the code nothing changes…why not?

WARNING: currently the code is only physically reasonable when the balls of equal mass start from rest OR if they do not collide. Also, the constants have been scaled to make the visualization more fun. Ways to vary the code to make it more physically reasonable under a wider variety of circumstances will be discussed on the pages following the code. **If you want to get more realistic collisions,** consider the last page of this handout and to see how things get pain-tastic in a hurry…

FYI – A fun wave code is at the very end of this handout.

Superballs in Space...

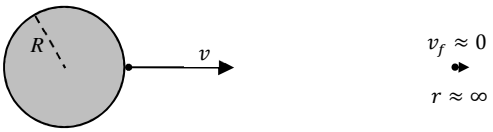
```
from visual import *
m1 = 8
m2 = 2
p1 = vector(0, 0, 0)
p2 = vector(0, 0, 0)
r = vector( 0, 0, 0)
deltat = 0.1
G = 6.67
t = 0

rock2 = sphere(pos=vector (10, 0, 0),
               color = color.orange,
               radius =0.8,
               make_trail = True,
               trail_type="points",
               interval=5,
               retain=5)

rock1 = sphere(pos=vector(-5, 0, 0),
               color = color.cyan,
               radius =0.2,
               make_trail=True,
               trail_type="points",
               interval=5,
               retain=5)

while t<200:
    rate(20)
    r = rock1.pos-rock2.pos
    if mag(r) <= rock2.radius + rock1.radius:
        p1 = -p1
        p2 = -p2
        rock1.pos = rock1.pos + (p1/m1) *deltat
        rock2.pos = rock2.pos + (p2/m2) *deltat
    else:
        rhat = r / mag(r)
        Fmag = G * m1 * m2 / mag(r)**2
        F_on1 = Fmag * (-rhat)
        F_on2 = -F_on1
        p1 = p1 + F_on1 * deltat
        rock1.pos = rock1.pos + (p1/m1) *deltat
        p2 = p2 + F_on2 * deltat
        rock2.pos = rock2.pos + (p2/m2) *deltat
    t = t + deltat
```

Are you still there? Feeling frisky I see…

1) The constant G is called the universal gravitation constant. The actual value is $G = 6.67 \times 10^{-11}\ \frac{\text{N}\cdot\text{m}^2}{\text{kg}^2}$.
   You may have noticed I used the number 6.67 in the simulation. If we use the actual value of G, what will happen to the forces in the simulation? Will the balls move faster or slower? A lot faster or a lot slower? Take a guess then run the sim to see if you are correct.
2) Assume we want to scale the masses appropriately such that the forces are unchanged as we scale G to the actual value. Assume we want to scale each mass by the same factor. By what factor must we scale the each mass?
3) FYI - To scale G to the correct factor of $G = 6.67 \times 10^{-11}\ \frac{\text{N}\cdot\text{m}^2}{\text{kg}^2}$ we *multiplied* G by 10^{-11}. Therefore, one must *divide* each mass by *the square root* of 10^{-11}.
4) Often, when doing orbit problems, we assume one mass is much bigger than the other. We do this so we can assume this mass remains stationary. Try playing with the masses until the big one remains almost perfectly at rest. Try mass ratios of 10:1, 100:1 and 1000:1. You may want to tweak the initial positions and radii of the balls so the simulation is a bit bigger. Try using the smaller ball with radius 1 and the bigger ball with radius 2. To make life easier for the next problem, put the bigger ball at the origin (0, 0, 0) and the smaller baller at (50, 0, 0).
5) For this part I will assume you are using a 1000:1 mass ratio with the big ball at the origin and the smaller ball is 50 units away. With such a large mass ratio, the large ball is essentially at rest. When this is the case, we can try to figure out the velocity needed for a circular orbit!

$$F_G = \frac{m_{small}v^2}{r}$$
$$\frac{Gm_{small}m_{big}}{r^2} = \frac{m_{small}v^2}{r}$$
$$v = ???$$

   Use the parameters of your code to determine the best value of v to get a circular orbit. Try it out! Think: which component (or components) should have this velocity? Do some trial and error until you get the circular orbit. Note: using $G = 6.67$ (*without* the 10^{-11}) my answer was close to $v = 11$…Note: when I ran my code the orbit didn't fit on the screen so glowscript automatically zoomed out as the code ran.
   Think: what if the speed is a little too big or too small? I suspect you will get elliptical orbits…but which axis should be the long axis (semi-major)? Take a guess for fun then try it.
   Going further: what if the mass ratio is small and the big mass moves. See the next page…
6) **Escape velocity** is discussed in problem **13.9** of my workbook. The planet is mass M and radius R. Mass m is launched *from the surface* at speed v. To escape m must attain large altitude ($r \approx \infty$) while still having at least a tiny bit of speed ($v_f \approx 0$).



$$U_{Gi} + K_i = U_{Gf} + K_f$$
$$-\frac{GmM}{R} + \frac{1}{2}mv^2 \approx -\frac{GmM}{\infty} + 0$$

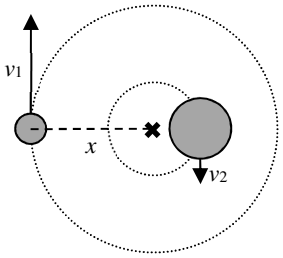
   Solve for v to determine the escae velocity of the smaller mass. Use your code to determine a number. WATCH OUT: the centers of the balls are initially separated by the *sum* of the radii!!! I assumed a 1000:1 mass ratio with the big ball of radius 2 at the origin and the smaller ball of radius 1 initially 3.01 units away. Also, zoom out in the early stages of the sim to make the visualization less confusing. Note: try it with 3.0 units away and see if you can figure out why the simulation fails. Using $G = 6.67$ (*without* the 10^{-11}) I found $v \approx 66.7$ *should* work. I had to play with the numbers a bit in the simulation on this one. In practice, $v \approx 70$ worked. Why is it off by so much? Play with the time step (`deltat`), rate, & total simulation time to see if a smaller time step makes the simulation more accurate…this worked for me. Also, try launching tangent to the surface just for fun! Try a tangential launch with a velocity slightly smaller than the escape velocity to see orbital precession…web search for "precession of Mercury".

7) Now imagine you have three rocks instead of just two. How do you modify the code? Do it.
8) The Coulomb force between two charged balls is very similar to the gravitational force.

Gravitational	Coulomb
$\vec{F}_{1on2} = \frac{Gm_1m_2}{r_{12}^2}(-\hat{r}_{1to2})$	$\vec{F}_{1on2} = \frac{kq_1q_2}{r_{12}^2}(\hat{r}_{1to2})$

The Coulomb constant is $k = 8.99\frac{\text{N}\cdot\text{m}^2}{\text{C}^2}$ and the charges q_1 and q_2 are given in Coulombs (C).
Try to modify the code to handle electrical forces instead of gravitational forces.

Reduced mass



In all of the previous orbit problems, we have been assuming the mass *m* of the satellite is much smaller than the central mass *M*. If this assumption is not valid, the central mass cannot be considered as at rest. The radius of circular motion is not the center-to-center distance *R*. Both *M* and *m* will be in circular orbits about the center of mass! Since the two objects are orbiting with different radii (and speeds) it will be less confusing to use $a_c = r\omega^2$instead of $a_c = \frac{v^2}{r}$. The two must have the same ω. Think: the two masses are always opposite each other on the circle, they must have the same rotation rate.

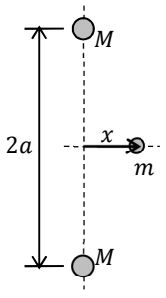
a) Show the distance *x* from the center of the *m* to the center of mass of the system is $x = R\frac{M}{(m+M)}$.
b) Write down the force equation for the circular motion of *m* using *R* as the center-to-center distance but *x* as the radius of circular motion. You should find

$$\frac{mM}{(m+M)}R\omega^2 = \mu R\omega^2 = \frac{GmM}{R^2}$$

where the *reduced mass* is defined as $\mu = \frac{mM}{m+M}$. Notice the force still uses the entire mass *m* while the acceleration is reduced using the reduced mass trick.
c) Assume the earth has mass 5.97×10^{24} kg and radius 6.37×10^6 m. The moon has mass 7.35×10^{22} kg and radius 1.74×10^6 m. The center-to-center distance from earth to moon is 3.84×10^8 m. Determine *x* and μ. I found $\mu = 7.35\times10^{22}$ kg and $x = 3.80\times10^8$ m. It is fun to notice the center of mass of the system is about 4×10^6 m from the center of the earth. This is approximately 70% of the earth's radius.

Note: a similar situation occurs an electron orbits a hydrogen nucleus. The spectra of light produced relates to the energies of the orbits. The reduced mass must be used to correctly predict the energies of orbits and the colors of light produced in the spectra! I find this interesting.

If you get a 3rd mass involved you could simulate this problem to find the effective spring constant:



13.14 Two objects of mass *M* are distance *a* above and below the origin as shown in the figure. A third mass *m* lies distance *x* to the right of the origin.

a) Determine the potential energy of the three mass system.
b) Determine the net gravitational force on the middle mass.
c) What value of *x* gives a maximum force? What is the maximum value of the force?
d) Assuming $x \ll a$, use the binomial expansion to determine an approximate equation for the force.
e) Plot the F_x vs *x* assuming $M = 10^{24}$ kg, $m = 1$ kg, and $a = 10^8$ m. Use values of x ranging from $-3a$ up to $+3a$. This is approximately like a 1kg satellite halfway between two earth's separated by the same distance as the earth and moon.

Notice if we keep only the first term in the expansion $\vec{F} = k_{eff}x(-\hat{\imath})$ where $k_{eff} = \frac{2GMm}{a^3}$. This implies the center mass should oscillate back and forth as if it were attached to a spring. Notice the plot of F_x vs *x* is linear, just like a spring, when $x \ll a$. Notice, however, the situation is unstable. If the mass moves slightly up or down the vertical forces no longer cancel out and the middle mass will be pulled towards the closer of the big masses.

What if you want to include arrows representing each force?

Remember this code:

```
ball=sphere(pos=vector(1, 2, 3),
            color=color.orange,
            radius=0.5)
velocity=vector(1, 3, 1)
arrow(pos=ball.pos,
        axis=velocity,
        color=color.green)
deltat=0.1
t=0
while t<80:
    rate(10)

    ball.pos=ball.pos+velocity*deltat
    t=t+deltat
```

If you recall, this code draws an arrow pointing in the direction of the velocity vector.

Adapt this code such that force vector arrows are draw to represent the gravitational force on each superball.

Graphs?

Perhaps you could add some energy bar graphs to your visualization somehow? Read a help file on the internet somewhere. The kinetic energy of a translating mass is given by $K = \frac{1}{2}mv^2$ while the gravitational potential energy is $U_G = -\frac{Gm_1m_2}{r_{12}^2}$. The total energy is just $E_{total} = K + U_G$. You can check your work verifying the total energy is constant for this simulation as the only force is gravitational between the balls. This force is thus internal and also conservative. System with no external or non-conservative work should conserve energy.

What about redoing projectiles and adding in air resistance?
You know how to model a ball experiencing force. Now consider the drag force. For many cases, the *magnitude* of the drag force is given by

$$R = bv^2$$

where v is speed and b is a constant.

This drag force points opposite the direction of motion. The direction of motion is simply a unit vector pointing in the direction of velocity. Therefore the vector drag force is given by

$$\vec{R} = bv^2(-\hat{v}) = -bv\vec{v}$$

Remember, the unit vector $\hat{v}$ is given by $\hat{v} = \frac{\vec{v}}{v}$.

The other force acting on a projectile after launch is

$$weight = -mg\hat{k}$$

You should be able make a sphere with some initial position and velocity. Then update the position and momentum following the procedure outlined on the first page of this handout.

To make it look cool, perhaps you could figure out how to add in some ground by calling the `box()` function. If you have non-zero initial launch altitude, perhaps a second box function could be called so you can see the ball launch off the top of a box and impact the box which is the ground. Then make the code stop running whne the ball impacts the ground.

A fun easy variation would be to launch one ball with air resistance and another ball without. Be sure to include trails as was done in the gravity code. Then you can compare the two like a race using your simulation. For fun, try launching both balls vertically with initial height of 19.6 m and initial velocity of about 19.6 m/s. Before hitting run, ask yourself which ball should hit the ground first? Then run the simulation and see if the answer is what you expected…it might surprise you.

Compare your code to a professional made simulation by performing a websearch for "Projectile Phet". Try to incorporate some of the features they used…see if you can reverse engineer it.

What about making a ball in box bouncing around each time it hits the walls? This is the start of a code to simulate molecules of a gas…Assume gravitational forces are negligible. To keep it simple, start with a ball which has initial momentum bouncing between two walls left and right. Start by keeping y- and z-compenets of momentum set to zero. Make walls using the `box()` function. Assume the ball reverse momentum each time it hits a wall. Then try a 2D initial momentum. Each time it hits a wall the component of momentum parallel to the wall will remain unchanged while the component perpendicular to the wall flips sign. Now try a 3D initial momentum vector. Then add in a second molecule. Go back to 1D first and make sure your code handles collisions of either ball with a wall AND the collision between the two balls. Now try to get all those things to work in 2D. Then in 3D (see next page for help on collisions between balls in 3D). Then introduce a 3rd ball. Then 100 balls. Let me know if you get it!

What if the collisions are not head on? Strictly speaking, sometimes the balls will not hit each other head on. Some collisions happen at a glancing angle.

The component of velocity *parallel* to $\hat{r}$ should be affected by the collision but the component *perpendicular* should be unaffected. If the $r \leq rock1.radius + rock2.radius$ one should split each momentum into two parts. One way to split up the momentum is to say

$$\vec{p}_2 = (a\ part \parallel to\ \hat{r}) + (the\ rest\ of\ it)$$
$$\vec{p}_2 = (\vec{p}_2 \cdot \hat{r})\hat{r} + (\vec{\boldsymbol{p}}_{\boldsymbol{2}} - (\vec{\boldsymbol{p}}_{\boldsymbol{2}} \cdot \hat{\boldsymbol{r}})\hat{\boldsymbol{r}})$$

Note: I defined $\hat{r}$ as $rock1.pos - rock2.pos$ so $\hat{r}$ points from rock2 (the initial pos) to rock1 (the final pos).

$$\vec{p}_1 = (a\ part \parallel to\ \hat{r}) + (the\ rest\ of\ it)$$
$$\vec{p}_1 = (\vec{p}_1 \cdot \hat{r})\hat{r} + (\vec{\boldsymbol{p}}_{\boldsymbol{1}} - (\vec{\boldsymbol{p}}_{\boldsymbol{1}} \cdot \hat{\boldsymbol{r}})\hat{\boldsymbol{r}})$$

Note: remember $\hat{r}$ points from rock2 to rock1 and $\vec{p}_1$ points from 1 to 2. Notice $\vec{p}_1 \cdot \hat{r}$ will be a negative number. As such this term implies rock1 has momentum pointing opposite $\hat{r}$…which is from 1 to 2 as it should be.

In addition, conservation of momentum should be used to determine the final momentum of each ball for the components parallel to $\hat{r}$.

$$\vec{p}_{2i} \cdot \hat{r} + \vec{p}_{1i} \cdot \hat{r} = \vec{p}_{2f} \cdot \hat{r} + \vec{p}_{1f} \cdot \hat{r}$$

Finally, if we assume *elastic* collisions, we can also use conservation of energy during the collisions. A convenient way to write translational kinetic energy from the momentum vector is

$$KE = \frac{1}{2}mv^2 = \frac{m^2v^2}{2m} = \frac{p^2}{2m} = \frac{\vec{p} \cdot \vec{p}}{2m}$$

Therefore conservation of energy during the collision states

$$\frac{\vec{p}_{1i} \cdot \vec{p}_{1i}}{2m_1} + \frac{\vec{p}_{2i} \cdot \vec{p}_{2i}}{2m_2} = \frac{\vec{p}_{1f} \cdot \vec{p}_{1f}}{2m_1} + \frac{\vec{p}_{2f} \cdot \vec{p}_{2f}}{2m_2}$$

In theory, you could use all this crap to make things bounce off at the correct angle and speed in 3D…no easy task if you ask me. That said, it simply requires a lot of patience and some coding. Once you get it right, you could in theory use this code in other problems involving collisions as well.

What if the collisions aren't elastic? The coefficient of restitution (e) is defined as

$$e = COR = \sqrt{\frac{KE_{after}}{KE_{before}}}$$

If you want you could use this COR to decrease the velocity of the ball after each impact. Instead of simply reversing the momentum of each ball after impact, each ball should reverse direction and slow down slightly. Equations which govern this slow down are

$$\vec{v}_{1after} = \frac{m_1\vec{v}_{1before} + m_2\vec{v}_{2before} + m_2e\left(\vec{v}_{2before} - \vec{v}_{1before}\right)}{m_1 + m_2}$$
$$\vec{v}_{2after} = \frac{m_1\vec{v}_{1before} + m_2\vec{v}_{2before} + m_2e\left(\vec{v}_{1before} - \vec{v}_{2before}\right)}{m_1 + m_2}$$

Note: don't forget the velocity relates to momentum using $\vec{p}_1 = m_1\vec{v}_1$ and $\vec{p}_2 = m_2\vec{v}_2$.
In theory you could use these equations in the if portion of the if/else statement to cause the balls to reverse direction and slowdown. You could add a constant such as COR = # where the # is something between 0 and 1.

- Notice a COR of 1 implies an elastic collision (energy before and after collision is the same).
- Notice a COR of 0 implies both objects have the same final velocity. This is a *perfectly* inelastic collision.
- A COR between 0 and 1 is called an inelastic collision.

An easy subset of inelastic collisions is *perfectly* inelastic. Perhaps you could write a could wherein any time the two balls touch they become one ball with a slightly larger radius that moves off with the correct momentum?

Wave Pulse On a String Code

```
from visual import *
A = 30
lambda = 15
k = 2*pi/lambda
omega = 4
i=-100
rope = []

while i<=100:
    vert_pos= A/(1+(k*(i+100))**2)
    ball = sphere(pos=vector(i, vert_pos, 0), radius=0.7, color=color.yellow)
    rope.append(ball)
    i=i+1

t=0
delta_t=0.1
j=-100
while t<=100:
    rate(20)
    while j<=100:
        vert_pos=A/(1+(k*(j+100)-omega*t)**2)
        rope[j].pos=vec(j, vert_pos, 0)
        j=j+1
    j=-100
    t=t+delta_t
```

To see if you really understand the code, try adding in a second wave pulse going the other direction.
Try messing with the time step to see when the code breaks down.
Think: try messing with the "wavelength" lambda and see when the pulse breaks down. Think how you might fix it.

Mass on a Spring Simulation:

```
from visual import*
H = 10
roof = box(pos=vec(0,H,0), length=10, height=0.2, width=10, color=color.cyan)
m = 5
k = 10
g = 10
eq_y = H-m*g/k
offset = 0.75*eq_y  #to be realistic, ensure offset < eq_y
#ground = box(pos=vec(0,-H-0.2,0), length=10, height=0.2, width=10,
color=color.green)

r = vec(0, eq_y+offset, 0)
delta_r = roof.pos - r
eq_line = curve(pos=[vec(-5, eq_y,0), vec(5, eq_y,0)])
ball = sphere(pos=r, radius=0.5)
spring = helix(pos=roof.pos, axis=-delta_r, radius=1)

scale = 0.1
F_s = k*delta_r
F_g = vec(0,-m*g, 0)
F_net = F_g + F_s
g_arrow = arrow(pos=ball.pos, axis=F_g*scale, color=color.red, opacity=0.7)
s_arrow = arrow(pos=ball.pos, axis=F_s*scale, color=color.blue, opacity=0.7)
net_arrow = arrow(pos=ball.pos, axis=F_net*scale, color=color.purple,
opacity=0.9)

p=vec(0,0,0)
t=0
delta_t = 0.02
while t<=100:
    rate(100)
    F_s = k*delta_r
    F_g = vec(0,-m*g, 0)
    F_net = F_g + F_s
    g_arrow.pos = ball.pos
    g_arrow.axis = F_g*scale
    s_arrow.pos = ball.pos
    s_arrow.axis = F_s*scale
    spring.axis = -delta_r
    net_arrow.pos = ball.pos
    net_arrow.axis = F_net*scale
    p = p + F_net*delta_t
    r = r + p/m*delta_t
    ball.pos = r
    delta_r = roof.pos - r
    t = t + delta_t
```

Projectile with and without air resistance code:

```
from visual import *
L = 125
H = 50
ground=box(pos=vec(0, -0.5, 0), color=color.green, length=L, height=0.5,
width=L)

speed = 25
theta = 30
g = 10
m = 10
b = 0.03    #this determines the drag coefficient

v_0 = vec(speed*cos(radians(theta)), speed*sin(radians(theta)), 0)
r_0 = vec(-L/2, H, 0)
F_grav = vec(0,-m*g, 0)

t = 0
delta_t = 0.1

p_1 = m*v_0
p_2 = p_1

pedestal = cylinder(pos=vec(r_0.x, 0, 0), axis=vec(0, H, 0),
color=color.green)
ball_1 = sphere(pos=r_0, color=color.red, make_trail=True)
ball_2 = sphere(pos=r_0, make_trail=True)

h_1=ball_1.pos.y
h_2=ball_1.pos.y

while h_1>=0:   #this is ball WITH air resistance
    rate(10)
    F_drag = -b/m**2*mag(p_1)*p_1
    F_net = F_grav + F_drag
    p_1 = p_1 + F_net*delta_t
    ball_1.pos = ball_1.pos + p_1/m*delta_t
    t = t + delta_t
    h_1=ball_1.pos.y

while h_2>=0:   #this is ball WITHOUT air resistance
    rate(10)
    p_2 = p_2 + F_grav*delta_t
    ball_2.pos = ball_2.pos + p_2/m*delta_t
    t = t + delta_t
    h_2=ball_2.pos.y
```

Balls connected by springs (needs some work still…see comments):

```
from visual import *
k = 200
m = 2
balls = []      #a list to be filled later
springs = []    #a list to be filled later
N = 10          #total number of balls to use
L_un = 5        #unstretched length of each spring

i = 0
while i<=N: #<= makes you have the extra ball at the end...see next loop
    ball = sphere(pos=vec(-0.5*(N*L_un)+i*L_un, 0, 0), radius=0.5,
color=color.red)
    balls.append(ball)  #this puts each ball in the list named balls
    i = i + 1

i = 0
while i<N:  #strictly < gives one less spring than balls
    spring = helix(pos=vec(-0.5*(N*L_un)+i*L_un, 0, 0), axis=vec(L_un, 0,0),
radius=0.5)
    springs.append(spring)  #this puts each helix in the list named springs
    i = i + 1

#I want to fix the leftmost and right most balls in place
#I will do this by making those masses really huge
M = 10000*m

#I will now take the second ball and shift it over a little bit
balls[1].pos.x=balls[1].pos.x+L_un/2

#I will now make a list of each ball's momentum
#and set it to zero so all balls start from rest
i = 0
p = vec(0, 0, 0)
moms = []
while i<= N:
    moms.append(p)
    i = i + 1

t = 0
delta_t = 0.0001    #be careful...if timestep too large, simulation goes
wacky!
while t < 100:
    rate(5000)
    j = 0
    while j < (N-1):
        #still needs some if/else statements
        #to accomodate the first and last balls having M not m
        #also need to account for masses only connected to a single spring
```

```
    #once done, should be able to modify code
    #to handle transverse or longitudinal waves
    #if you initialize the sphere positions based on a wave equation
    #should be able to see the compression propogate like a wave
    springs[j].pos = balls[j].pos
    springs[j].axis = balls[j+1].pos - balls[j].pos
    F_L = vec(-k*((balls[j+1].pos.x - balls[j].pos.x)-L_un), 0, 0)
    F_R = vec(+k*((balls[j+2].pos.x - balls[j+1].pos.x)-L_un), 0, 0)
    F_net = F_L + F_R
    moms[j+1] = moms[j+1] + F_net*delta_t
    balls[j+1].pos = balls[j+1].pos + 1/m*moms[j+1]*delta_t
    j = j + 1
t = t + delta_t
```