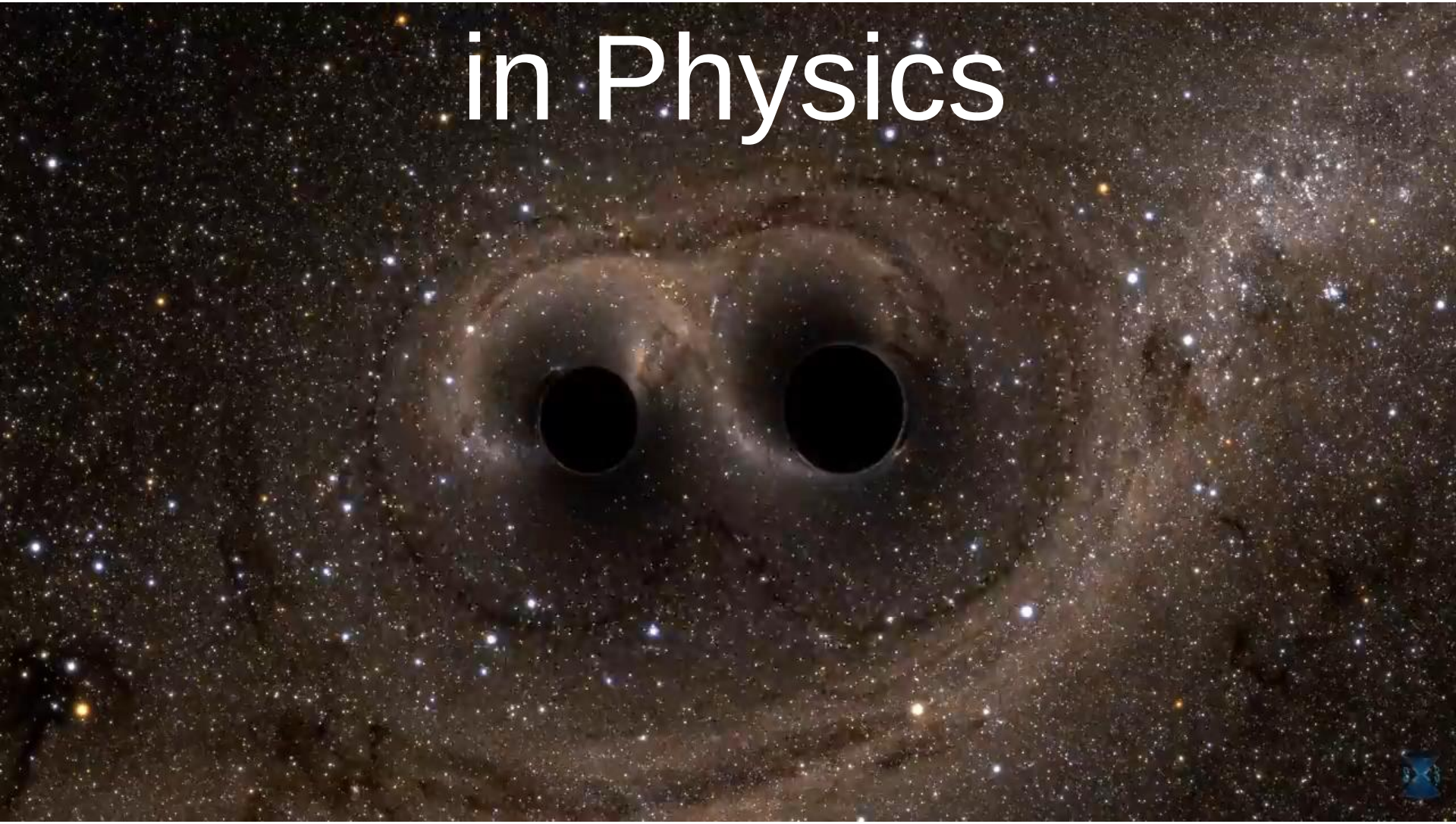
# Scientific Computing in Physics

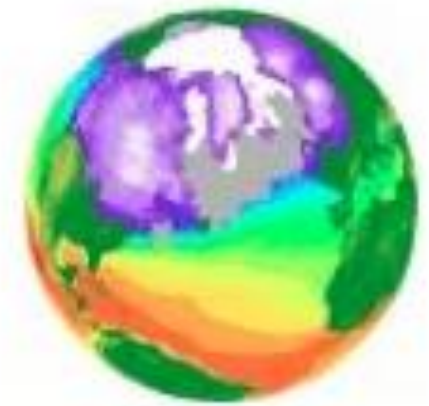


August 15th
SUSC, Physics Department

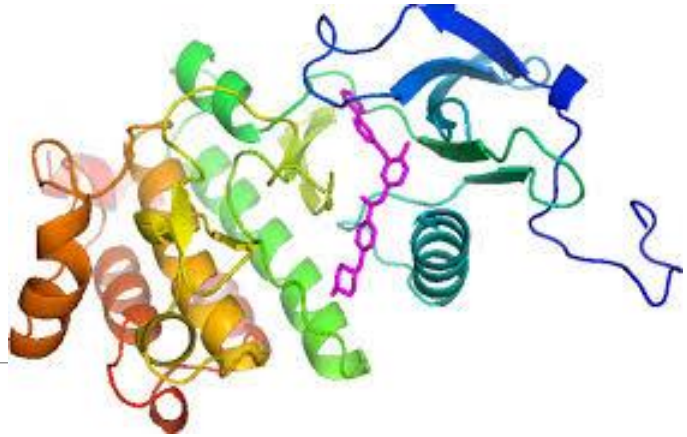
CJ Woodford
Canadian Institute of Theoretical Astrophysics

Physics
UNIVERSITY OF TORONTO

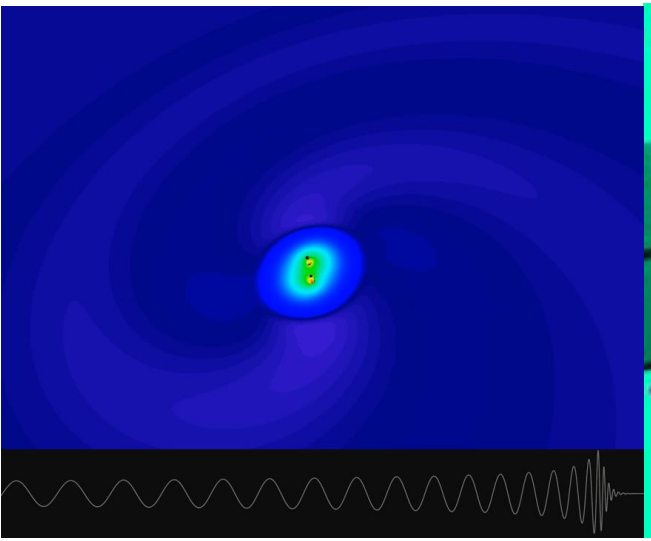
CITA | ICAT

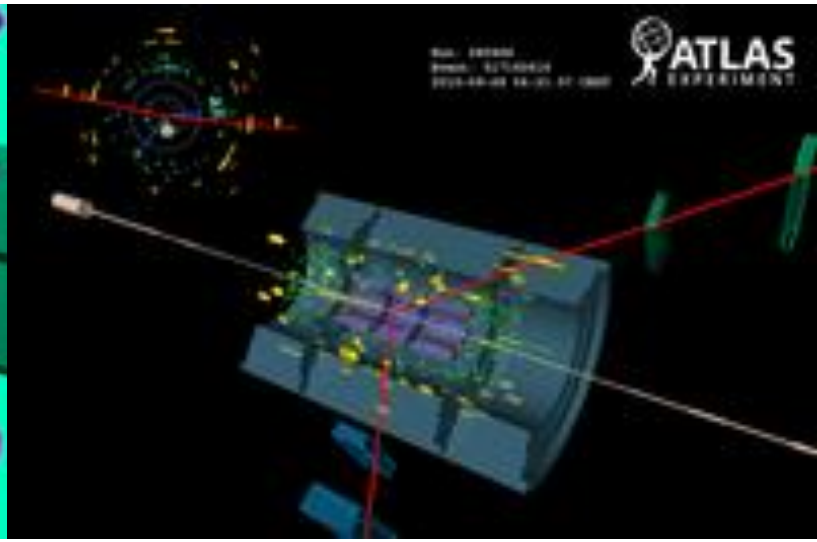


# What is Scientific Computing?



Using computers to simulate real life scenarios, in order to forecast, estimate, or constrain real-world problems.

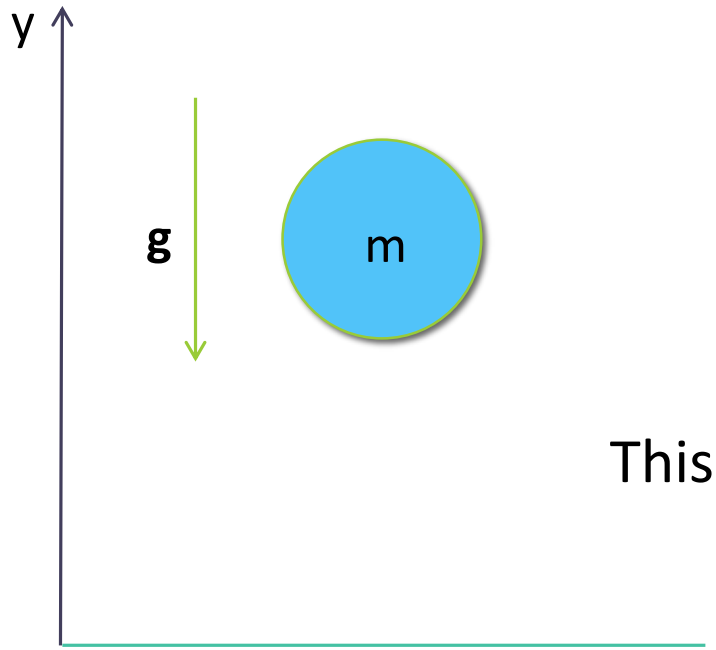




# Why Physics Uses Computing

- Some physics problems have simple equations that can be solved "analytically" (i.e. with pen and paper). These usually involve simple forces and few objects being acted upon:

- e.g.: ball under the influence of gravity: $F = ma$
$= -mg$




**Solution:**

velocity: $v_f = v_i - g(t_f - t_i)$

position: $y_f = y_i + v_i(t_f - t_i) - \frac{1}{2}g(t_f - t_i)^2$

This is a very IDEALIZED problem: Why?

# Why Physics Uses Computing

- Forces can be much more complicated:
  - e.g. fluid in Earth's core:

$$\vec{F} = -\nabla P + \rho \vec{g} + \mu \nabla^2 \vec{v} + \frac{1}{3}\nabla(\nabla \cdot \vec{v}) - 2\rho \vec{\Omega} \times \vec{v}$$

- System may include many components, each affected by different forces
  - e.g. Binary Neutron star merger, detected by LIGO and simulated at Goddard

- These problems can't be solved with pen & paper - Instead we need the rapid calculation power and large memory of the biggest computers available

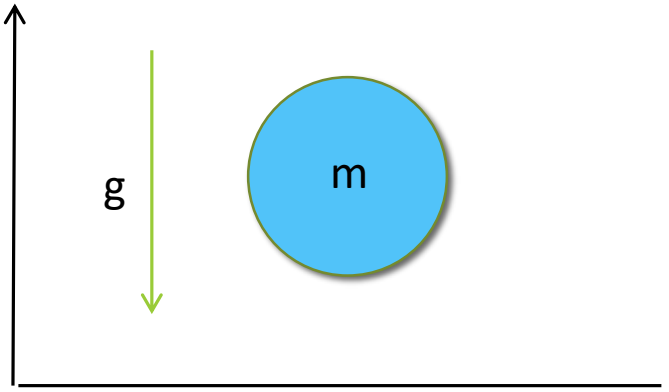
# How Does a Computer Solve Physics Problems?

- Computers are dumb, but very good at doing basic math fast!
- What we need to do:
  1. Find the equations that describe the physics
  2. Rewrite them in terms of simple math operations
     i. There may be MANY of these operations but that's okay because computers are fast! The fastest supercomputer in the world (Top 500 list) is Summit (or OLCF-4) – can do 122.3 PFLOPS
  3. Tell the computer to do the mathematical operations

# How Does a Computer Solve Physics Problems?

- Example: Ball under the influence of gravity:

$$F = ma$$
$$= -mg$$

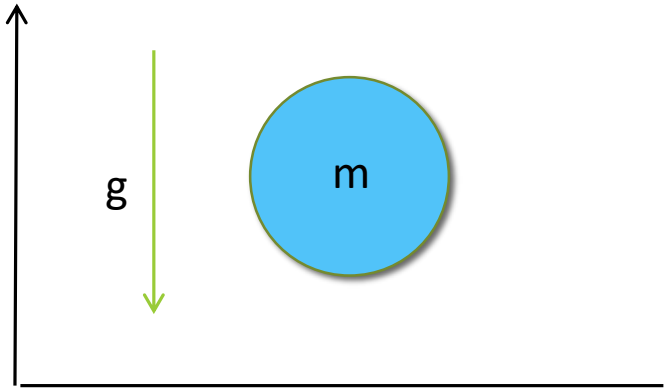



- Lets pretend we don't know the solution (since we don't for other problems). How do we get the computer to calculate the solution?

# How Does a Computer Solve Physics Problems?

- Example: Ball under the influence of gravity:

$$F = ma$$
$$= -mg$$




- Lets pretend we don't know the solution (since we won't for other problems). How do we get the computer to calculate the solution?

1. Determine the equations governing the physics we are studying: (these equations are true as long as $\Delta t$ is very very small)

Newton's 2nd Law:

$$F = ma \qquad a = \frac{\Delta v}{\Delta t}$$
$$= -mg \qquad v = \frac{\Delta y}{\Delta t}$$
$$a = \frac{F}{m}$$

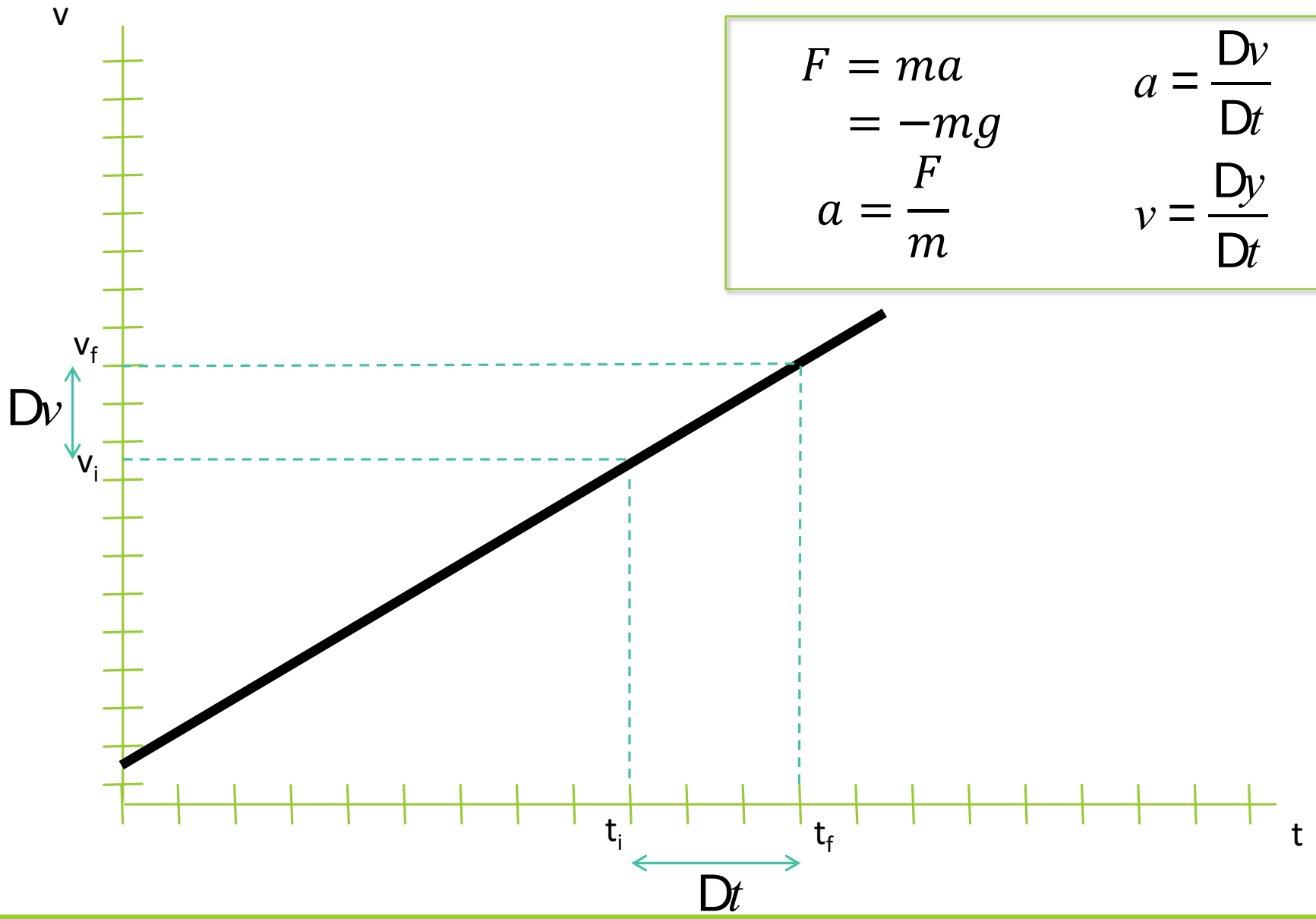
$\Delta t$ : small interval of time

$\Delta v$: change in velocity over $\Delta t$

$\Delta y$: change in position over $\Delta t$

# How Does a Computer Solve Physics Problems?

$$F = ma \qquad a = \frac{\Delta v}{\Delta t}$$
$$= -mg$$
$$a = \frac{F}{m} \qquad v = \frac{\Delta y}{\Delta t}$$



# How Does a Computer Solve Physics Problems?

2. Rewrite them in terms of simple mathematical operations
   ii. Rewrite our equations for a(t) and v(t) using:

$$\Delta v = v_f - v_i$$

$$\Delta y = y_f - y_i$$

$$F = ma \qquad a = \frac{\Delta v}{\Delta t}$$
$$= -mg \qquad v = \frac{\Delta y}{\Delta t}$$
$$a = \frac{F}{m}$$

We get:

$$v_f = v_i - g\Delta t$$
$$y_f = y_i + v_f \Delta t$$

These are now equations using basic math operations on how to update the velocity and position after each time step.

Notice these aren't the analytic solutions:

$$v_f = v_i - g(t_f - t_i)$$
$$y_f = y_i + v_i(t_f - t_i) - \frac{1}{2}g(t_f - t_i)^2$$

# How Does a Computer Solve Physics Problems?

3. Tell the computer to do the math!

$$v_f = v_i - g\Delta t$$
$$y_f = y_i + v_f \Delta t$$

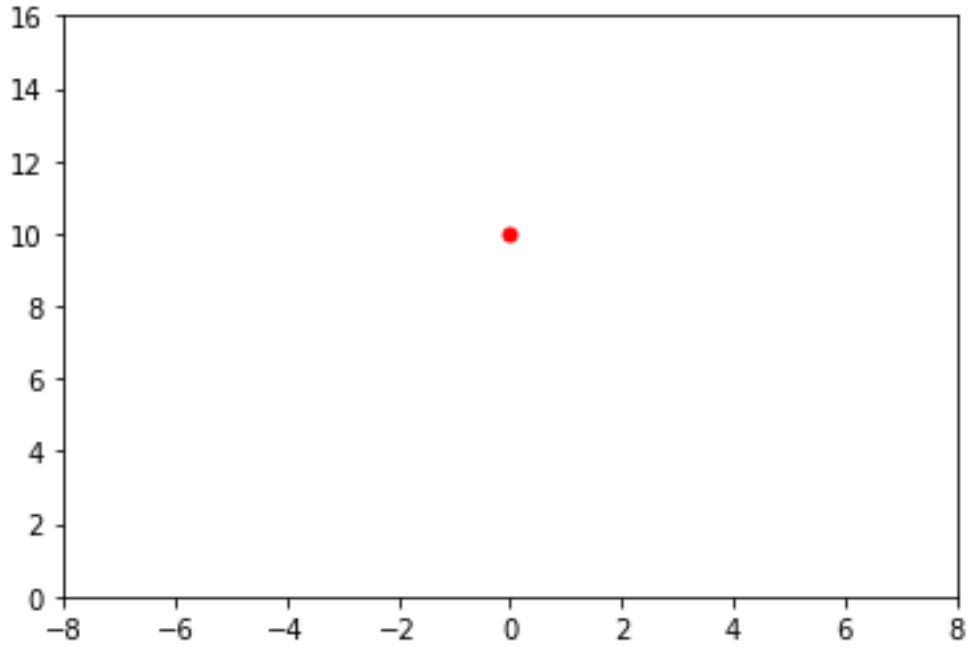
So our *algorithm* should be:

1. Start with t=0, $v_i$, $y_i$ defined at t=0.
2. Define $\Delta t$ (up to you!)
3. Take 1 step in time ($1\Delta t$)
4. Calculate new $v_f$ and $y_f$ from formulas above
5. Repeat steps 3 – 5 for however many time steps we need

Lets use the python programming language to do this!

# Free Fall Code

- Click on the “Spyder” icon on the desktop
- Under the “File” menu open “freefall.py” on the desktop
- Save as “freefall_v1.py” (in case you need to start over)
- Close the help window, so you only have the code and iPython console screens.
- Run your program using “F5” or the big green arrow in the tool bar.
- You should get the following in the iPython console:



# Free Fall Code

- Lets look at the program:
  - Any line starting with # is a comment, the code does not read it
  - This line defines the position of the ball in 2D:

```
ball = np.array([0.0,10.0])
```

x-position y-position

  - These lines define the ball's initial velocity and acceleration due to gravity:

```
ball_vel=np.array[(0.0,0.0)]
g=9.8
```

Since that is the only part of the code active right now, no time steps are taken so the ball never updates its velocity or position and stays at the same location (that's why nothing happens).

# Free Fall Code

- Okay, so lets make something happen:
  1. Start with t=0
  2. Define $\Delta t$
     - In the code, remove the # sign in front of the following lines:

```
t=0.0
dt = 0.01
```

  3. Take 1 step in time (1 $\Delta t$)
  4. Calculate new $v_f$ and $y_f$

- In the code, remove the # sign in front of the following lines (make sure to keep the indentation):

```
t=t+dt
ball_vel[1] = ball_vel[1] - g*dt
ball[1] = ball[1] + ball_vel[1]*dt
```

$\longleftrightarrow$

$$v_f = v_i - g\Delta t$$

$$y_f = y_i + v_f \Delta t$$

# Free Fall Code

- Repeat steps 3 – 4 for however many time steps we need
  - The best way to do this with a computer is using a “loop”. A “while” loop repeats all the commands indented in the lines after the while statement as long as some condition is met. There are other kinds of loops you can try in your own codes, like “for” loops.
  - Remove the # sign in front of the line: `while ball[1] >= 0.0` what does this mean?
  - This starts our loop. The commands indented afterwards:

```
t=t+dt
ball_vel[1] = ball_vel[1] - g*dt
ball[1] = ball[1] + ball_vel[1]*dt
```

  will be calculated each time through the loop.

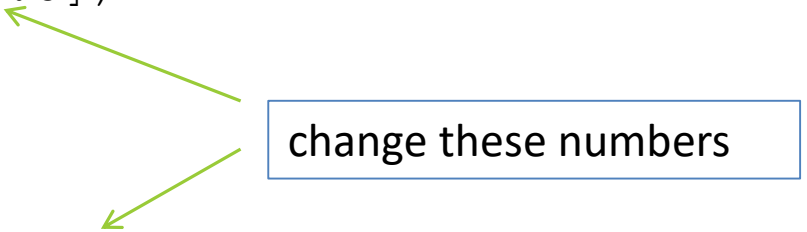
  - The loop will keep repeating until the position of the ball < 0 (the ball hits the ground).
  - Uncomment the rest of the loop and the last line to include graphic output. Run your program.

# Free Fall Code

- Try running your program with different initial y velocities and different initial y positions by changing the following lines:

```
ball = np.array([0.0,10.0])
```

```
ball_vel = np.array([0.0,0.0])
```




- What changes for your runs? Is there a difference for positive and negative initial velocity values?

# Projectile Motion Code

- Open the program “projectile.py”. Save it as “projectile_v1.py”.
- Notice that currently it is almost identical to the free fall code. If you run it you will see it is the same.
- Projectile motion also allows initial velocities in the “x” (horizontal) direction so instead of dropping the ball we will throw the ball.
- Your mission: Modify some of the lines and add lines in this code to give the ball an initial velocity in the x direction and to update the x components of the position and velocity in time.
- ***If you cannot see the ball on the plots, it means that it’s position is outside of the frame boundary. Change the values in the lines to larger/smaller values.

```
plt.axis([-16,16,0,16])
```

x-axis min and max  y-axis min and max

# Projectile Motion Code

- Is there any acceleration in the x direction?

# Projectile Motion Code

- Is there any acceleration in the x direction?

No, gravity only works in the y direction. So in the x direction:

$$v_f = v_i$$
$$x_f = x_i + v_f \Delta t$$

- Do we ever need to update the velocity in the x direction?

# Projectile Motion Code

- Is there any acceleration in the x direction?

  No, gravity only works in the y direction. So in the x direction:

$$v_f = v_i$$
$$x_f = x_i + v_f \Delta t$$

- Do we ever need to update the velocity in the x direction?

  No, its always the same. It's a constant = initial x velocity.

- Do we need to update the position in the x direction?

# Projectile Motion Code

- Is there any acceleration in the x direction?

  No, gravity only works in the y direction. So in the x direction:

$$v_f = v_i$$
$$x_f = x_i + v_f \Delta t$$

- Do we ever need to update the velocity in the x direction?

  No, its always the same. It's a constant = initial x velocity.

- Do we need to update the position in the x direction?

  Yes! According to the above formula.

# Projectile Motion Code

- Here is the line in the code for the y position.

```
ball[1] = ball[1] + ball_vel[1]*dt
```

- Create a very similar line for the "x" position (replace all the 1's with 0's) and add it to the code right after the above line. Don't forget to keep the line indented!
- Run your code. Did you get projectile motion?

- Try varying your initial x and y velocities to see what you get.

- You have now created a projectile motion code!

Recap:

- We defined the laws of physics for our object
- We rewrote the laws into simple mathematical operations
- We made the computer do the operations over and over again to update the motion of the object.

# Contact Forces

- Gravity is an easy force to work with because it acts at all times and on all of the body

- What about modeling another type of force? For example:

- Contact force due to a collision: Very hard to model! But luckily we have some physics "conservation" laws to help us.

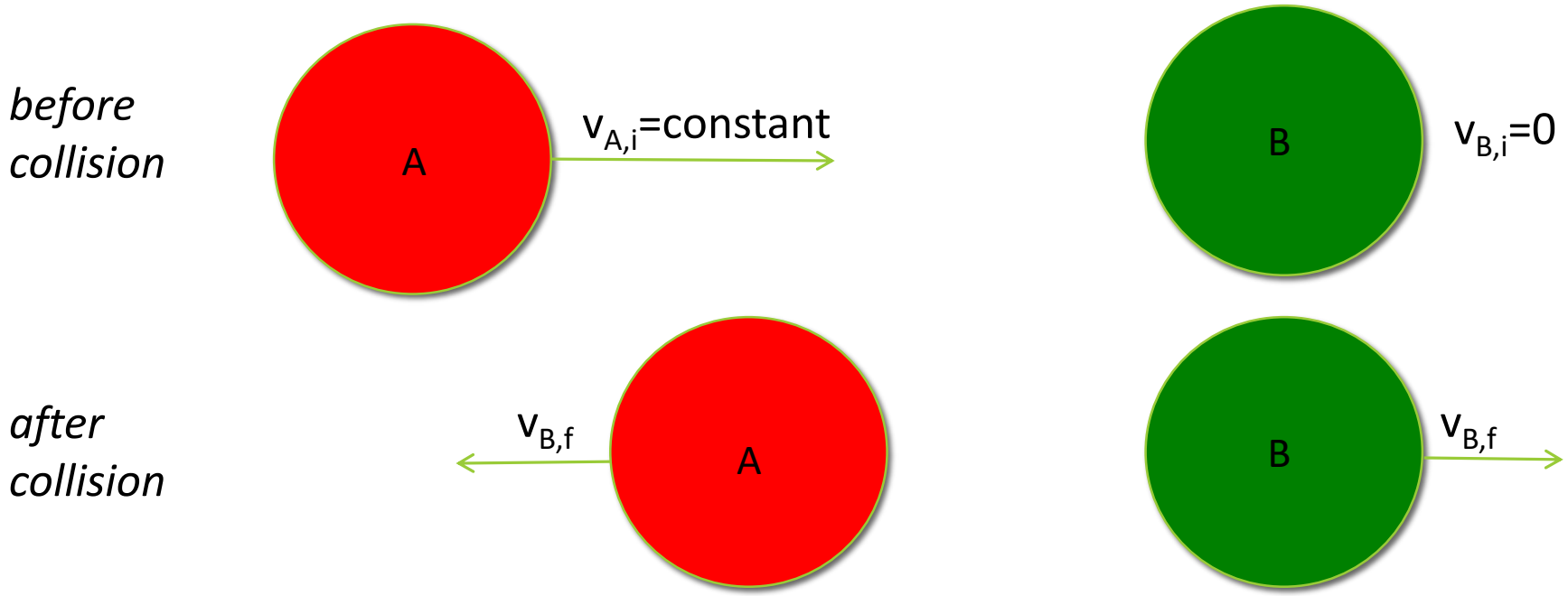

What happens when they collide?

# Physics of Collisions

It depends!

- What are the balls made of?
  e.g. Tennis balls collide differently than balls made of peanut butter (try this at home!)

- We will consider a special kind of collision: an "elastic collision". In this case, the energy is conserved in the collision



# Physics of Collisions

- Use momentum conservation and energy conservation laws to determine final velocities:

Momentum = mass * velocity

$$\rho = m * v$$

Kinetic Energy = 0.5* mass * velocity^2

$$\mathrm{E} = \frac{1}{2} m * v^2$$

Momentum Conservation Law: Consider all objects in system. If no outside forces acting, then total momentum of system stays constant even if the objects collide.

Energy Conservation Law: The total energy of the system must also be conserved. Here we only consider kinetic energy and no external forces.

$$\rho_{total} = m_1 * v_1 + m_2 * v_2,$$
$$E_{total} = \frac{1}{2} m_1 * v_1^2 + \frac{1}{2} m_2 * v_2^2$$

are constant in time

→ Initial momentum (before collision) = Final momentum (after collision)

→ Initial Energy (before collision) = Final Energy (after collision)

# Physics of Collisions

Momentum conservation: $m_A v_{A,i} + m_B v_{B,i} = m_A v_{A,f} + m_B v_{B,f}$

Energy conservation for an "elastic" collision: $\frac{1}{2} m_A v_{A,i}^2 + \frac{1}{2} m_B v_{B,i}^2 = \frac{1}{2} m_A v_{A,f}^2 + \frac{1}{2} m_B v_{B,f}^2$

We will only consider collisions where $v_{B,i} = 0$ (it makes the math easier). Then we can solve the above two equations for the final velocities after the collision:

$$v_{B,f} = \frac{2m_A}{(m_A + m_B)} v_{A,i} \qquad v_{A,f} = v_{A,i} - \frac{m_B}{m_A} v_{B,f}$$

Lets work on a code to model these elastic collisions with the above equation.

Open "collision.py". Save as "collision_v1.py"

# Collisions Code

- Code creates 2 balls (ballA & ballB):

```
ballA= np.array([-4.0,0.0])
ballB= np.array([2.0,0.0])
```

- sets the mass of each ball:

```
mA=1.0
mB=4.0
```

- defines the balls' initial velocities:

```
ballA_vel= np.array([2.0,0.0])
ballB_vel= np.array([0.0,0.0])
```

- then updates the positions of the balls using a while loop:

```
while 1==1:
   ballA = ballA + ballA_vel*dt
   ballB = ballB + ballB_vel*dt
```

- Run the program, what happens?

# Collisions Code

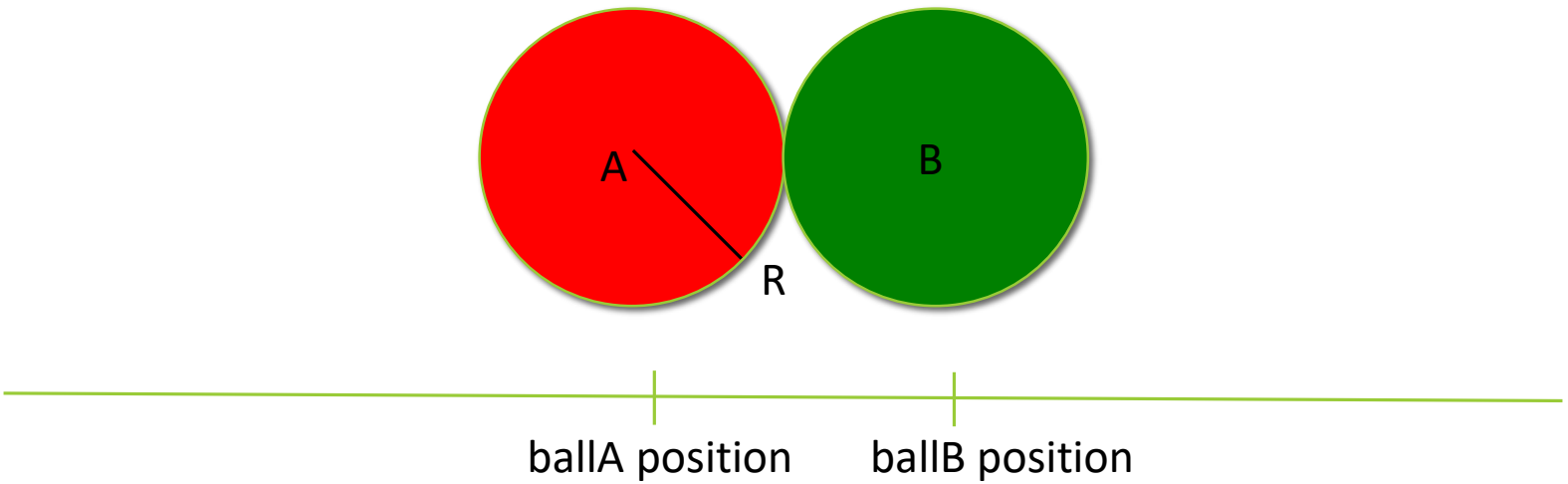
- The balls don't collide! Well that's because we haven't told the computer about the collision. (COMPUTERS ARE DUMB!)

- To model the collision we need to tell the computer :
  1. how to determine if a collision has occurred
  2. how to calculate the velocities after the collision

Determining when a collision occurs in terms of things the computer knows:


Think about a condition on the positions of the balls that would mean a collision occurs.

# Collisions Code




```
ballB[0] - ballA[0] = rA + rB
```

- We only want to change the velocities if a collision occurs. So we will use an “if” statement to determine this
- In the “while loop” remove the # in front of the line:

```
if ?????:
```

# Collisions Code

```
if ?????:
```

- Replace the ????? with:

```
ballB[0] - ballA[0] <= rA + rB
```

***Make sure to leave the " : " after the question marks***
***Notice the " <= " (this is because computers aren't good with equals) ***

- Now lets put in the equations for the velocities after the collision:

$$v_{B,f} = \frac{2m_A}{(m_A + m_B)} v_{A,i} \qquad v_{A,f} = v_{A,i} - \frac{m_B}{m_A} v_{B,f}$$

Uncomment the following lines of code (but keep the indentation):

```
ballB_vel=2.0*mA/(mA+mB)*ballA_vel
ballA_vel=ballA_vel-mB/mA*ballA_vel
```

- Run your program. What happens?

# Collisions Code

Your final new code lines (ignoring plotting lines) should look like:

```
while 1==1:
    ballA = ballA + ballA_vel*dt
    ballB = ballB + ballB_vel*dt

    if (ballB[0]-ballA[0] <= rA + rB:
       ballB_vel=2.0*mA/(mA+mB)*ballA_vel
       ballA_vel=ballA_vel-mB/mA*ballB_vel
```

- Try varying the initial velocity of ball A and masses of the balls to see what happens.

# Collisions Code

Recap:

- We defined the laws of physics for our collision
- We rewrote the laws into simple mathematical operations
- We made the computer do the operations over and over again to update the motion of the balls and feel collisions between them.

So we've seen how to model projectile motion and collisions.

That reminds me of a certain app…

# Angry Spheres Level 1

- Angry birds has lots of physics going on
- Energy conservation: Launch of Bird
- Projectile motion: Flight of Bird
- Collisions: Bird, pigs, blocks, …

So angry!

- Using what we've learned from our previous coding experiences, we could quickly come up with a basic "Angry Spheres" code:
- open the program AngrySpheres_L1.py and run it.
- You see a red sphere and a green sphere. The object of the game is to launch the "Angry" red sphere at some initial velocity and hit the green sphere.

# Angry Spheres Level 1

- The plan:
  - Give red sphere an initial velocity
  - Make red sphere undergo projectile motion
  - Make red sphere collide with green sphere

- Lets look through the code in AngrySpheres_L1.py:

```
#create the bird and pig
Rbird=0.5
Rpig=0.5
bird = np.array([-15.0,0.0])
pig = np.array([10.0,0.0])
```

- This bit creates the “bird” (red sphere) and “pig” (green sphere), just like we did for the balls in collisions.py and projectile.py

# Angry Spheres Level 1

```
#define the initial velocities and acceleration due to gravity
g=9.8
bird_vel = np.array([0.0,0.0])
pig_vel = np.array([0.0,0.0])
acc = np.array([0.0,-g])
```

- again, very similar to what we did in projectile.py

```
#masses
Mbird=1.0
Mpig=2.0

#set the initial time=0 and set the time step
t = 0.0
dt = 0.005
```

- masses are needed for the collision, time step and dt for the time evolution.

# Angry Spheres Level 1

```
#This "while" loop repeats the commands indented after it while
the condition is true
while 1 == 1:
    t=t+dt   #updates the time

    #update velocity and position of bird
    bird_vel = bird_vel + acc*dt    #v_f=v_i+a*dt
    bird = bird + bird_vel*dt    #y_f=y_i+v*dt

    #update position of pig
    pig_vel = pig_vel + acc*dt
    pig = pig + pig_vel*dt
```

- again similar to what we did in projectile.py for the bird
- now we also update the pig's position

# Angry Spheres Level 1

```
#condition for pig and bird to collide
    if abs(mag.norm(bird-pig))<= Rbird + Rpig:
        if birdpigcollided ==False: #only first time
            pig_vel=2.0*Mbird/(Mbird+Mpig)*bird_vel
            bird_vel=bird_vel-Mpig/Mbird*pig_vel
            birdpigcollided=True
```

- Similar to what we did in collision.py but 1 major difference:
    - We are allowing the spheres to move in 2D so the collision will involve velocities in 2 directions and the condition for collision has to involve the magnitude of the position vectors.

# Angry Spheres Level 1

```
#damping factor for collisions with floor
dampx=0.999
dampy=0.9

#Interaction with floor causes damped bouncing
    if bird[1] <=0.0:
        bird_vel[1]=-dampy*bird_vel[1]
        bird_vel[0]=dampx*bird_vel[0]
    if pig[1] <=0.0:
        pig[1]=0.0
        pig_vel[0]=dampx*pig_vel[0]
```

- This is new for us. We're now forcing the bird to "bounce" if it hits the floor (kind of like it does when it hits the pig). But I wanted the floor to dampen the motion.
- We also don't want the pig to fall through the floor so its y position is set to stay on the ground.

# Angry Spheres Level 1

- Set the initial velocity of the bird (any value you want) by altering the following line:

```
bird_vel = np.array([0.0,0.0])
```

- Use trial and error to find a velocity that will hit the pig.

- We can use physics laws to determine a velocity that will work…

# Angry Spheres Level 1

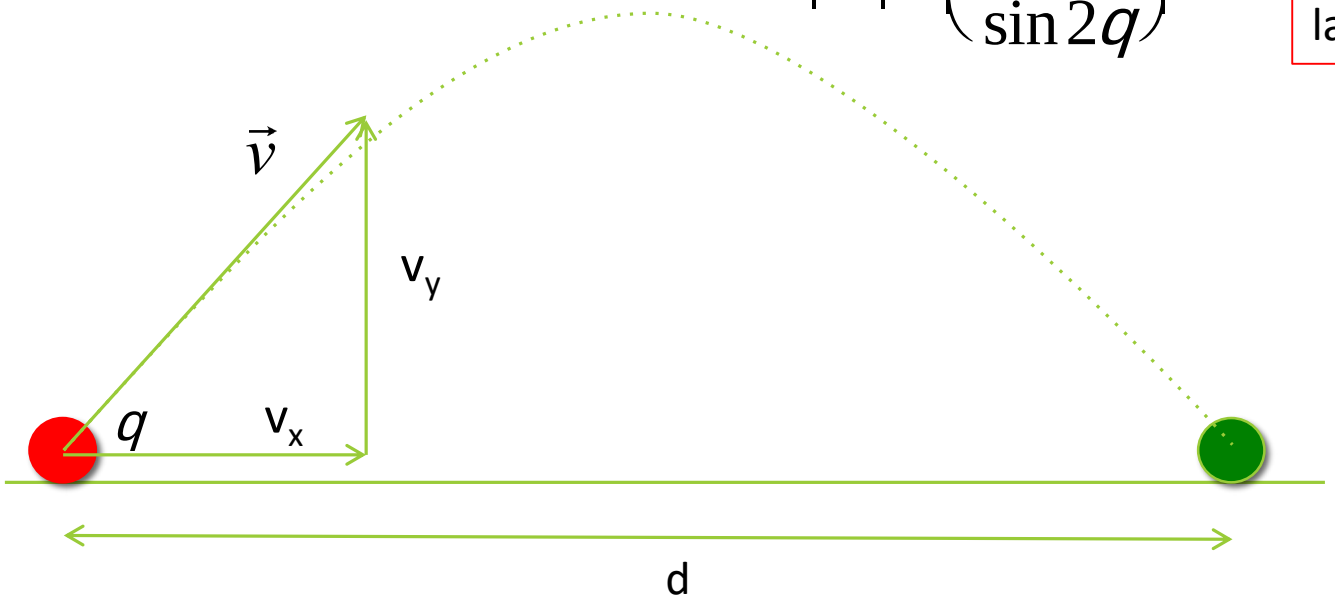
- Here is the formula. It can be derived from the projectile motion equations:

$$v_y = \frac{gd}{2v_x}$$

- Another way to think of this: given an angle, there is a specific magnitude of velocity that can be used to hit the pig.

$$|\vec{v}| = \left(\frac{gd}{\sin 2q}\right)^{0.5}$$

**remember this formula for later!!!



# Angry Spheres Level 1

- Lets check if our formula works by using it to set the initial velocity:

$$v_y = \frac{gd}{2v_x}$$

- Replace the line:

```
bird_vel = np.array([0.0,0.0])
```

- With:

```
vx = 5.0
vy = g*(pig[0]-bird[0])/(2.0*vx)
bird_vel = np.array([vx,vy])
```

- Run the code and see if it works.
- Then try other values for $v_x$ to see that it works for any of them.

# Angry Spheres Level 2

- What about introducing a block that can fall on the pig?
- Open AngrySpheres_L2.py and run it.
- create a “block” using the line:

```
#creating block
H=10.0
R=0.5
block = cylinder(pos=(5,-Rpig,0), axis=(0,H,0), radius=R,
                                                color=color.blue)
```

- define some initial parameters for the block’s rotation if its hit:

```
#define initial velocity and parameters for rotation of
block
block.vel=vector(0,0,0)
Iblock = Mblock*H**2/3.0 #moment of inertia
omega=0.0
theta=pi/2.0
```

# Angry Spheres Level 2

- update the block's rotation using laws of physics:

```
#update rotation of block
    alpha=-g*cos(theta)/H   #calculate the angular acceleration
    omega=omega+alpha*dt    #calculate the angular velocity
    theta=theta+omega*dt    #calculate the angle to determine
                            #the block's axis
    block.axis = H*vector(cos(theta),sin(theta),0)
```

- create the condition for the bird and block to collide:

```
#condition for bird and block to collide
    if bird.pos.x >= block.pos.x-Rbird-R and bird.pos.x <=
                            block.pos.x+Rbird+R and bird.pos.y <= H:
        if birdblockcollided == False:
            Ibird=Mbird*bird.pos.y**2
            omega=-2.0*Ibird/(Ibird+Iblock)*bird.vel.x/bird.pos.y
            bird.vel=bird.vel-Iblock/Ibird*block.vel #elastic collision
            bird.vel.x=-bird.vel.x
            birdblockcollided=True
```

# Angry Spheres Level 2

- create the condition for the block and pig to collide:

```
#condition for block and pig to collide
    if theta <= atan((2*Rpig+0.8*R)/(pig.pos.x-block.pos.x)):
        omega=0.0
        alpha=0.0
        label(pos=(0,16,0), text='SUCCESS!', height=48)
```

- create the condition for the block and ground to collide:

```
#condition for block to collide with ground if omega <0
    if theta >= pi:
        omega=0.0
        alpha=0.0
```

# Angry Spheres Level 2

- Try various initial bird velocities to see if you can hit the block and knock it onto the pig
- Try various initial bird velocities to see if you can hit the pig WITHOUT hitting the block
- Remember the formula:

$$v_y = \frac{gd}{2v_x}$$

# Resources:

1. Lynda online coding courses (www.lynda.com)
2. Ladies Learning Code (www.ladieslearningcode.com)
3. Enthought Canopy (FREE platform for using, creating, and running python code! – www.enthought.com/products/canopy/)
4. Python tutorials and walkthroughs (www.python.org)
5. Download and use Spyder yourself!
   i. Follow instructions on our wiki: http://compwiki.physics.utoronto.ca/ , in the Python 3 section
   ii. Download Anaconda 3 directly: https://www.continuum.io/downloads
6. Hatch (www.hatchcanada.com)

Feel free to contact me if you have any questions about this session!

cwoodford@black-holes.org

You can also find all these resources plus the code on my website: www.cita.utoronto.ca/~woodford