PHYS 566 HW3

Matthew Epland
Department of Physics, Duke University, Durham, NC 27707, USA
(Dated: February 11, 2016)

The trajectory of a typical golf ball was computed numerically via the Euler method for four cases: ideal motion, smooth drag, dimpled drag, and dimpled drag with spin. The computational method and code were successfully validated against the known parabolic behavior of the ideal case. The final case of a dimpled golf ball with backspin produced trajectories similar to those observed by real world golf balls in flight, demonstrating the power of computational methods.

I. INTRODUCTION

From its modern beginnings with Galileo, physics has always been interested in the trajectories of objects undergoing projectile motion. In the ideal case without drag, the object's trajectory can easily be derived analytically, as is done countless times each fall in introductory mechanics courses around the world. However, adding a simple drag force $\mathbf{F}_{\text{drag}}$ proportional to velocity v drastically complicates the problem. While still solvable analytically, the simple drag case pushes up against the bounds of what is reasonable to do with pencil and paper. Going any further by allowing $\mathbf{F}_{\text{drag}} \propto v$ or v^2 depending on the velocity regime, or by spinning the object, turns any analytical efforts into exercises in futility. Fortunately, all of these cases can be readily handled numerically with about the same amount of effort no matter how many drag or spin forces are considered. In this assignment the trajectory of a golf ball with mass $m = 0.046$ kg, launched at an angle of $9^\circ \leq \vartheta \leq 60^\circ$, with initial velocity $v_0 = 70$ m/s, was computed numerically via the Euler method. Four cases; ideal motion, smooth drag, dimpled drag, and dimpled drag with spin, based on those mentioned above and detailed in the subsequent section, were considered in turn.

II. THEORY

We can numerically model an objects trajectory by splitting up Newton's 2nd Law (1) into coupled first order differential equations (2) that we can apply the Euler method (3 and 4) to.

$$m\frac{d^2\mathbf{r}}{dt^2} = \mathbf{F}_{\text{net}} \tag{1}$$

$$\frac{d\mathbf{v}}{dt} = \frac{1}{m}\mathbf{F}_{\text{net}} \qquad \frac{d\mathbf{r}}{dt} = \mathbf{v} \tag{2}$$

$$\mathbf{r}\left(t+\Delta t\right) \approx \mathbf{r}\left(t\right) + \frac{d\mathbf{r}}{dt}\Delta t = \mathbf{r}\left(t\right) + \mathbf{v}\left(t\right)\Delta t \tag{3}$$

$$\mathbf{v}\left(t+\Delta t\right) \approx \mathbf{v}\left(t\right) + \frac{d\mathbf{v}}{dt}\Delta t = \mathbf{v}\left(t\right) + \frac{1}{m}\mathbf{F}_{\text{net}}\left(t\right)\Delta t \tag{4}$$

Once $\mathbf{F}_{\text{net}}$ is specified, these equations form the heart of our model. All we need do now is split the vectors up into components, $\hat{\mathbf{x}}$ horizontal along the ground and $\hat{\mathbf{y}}$ vertical, by performing some simple trigonometry (5) with ϑ defined as the angle from horizontal, and set the initial conditions (6).

$$\vartheta\left(t\right) = \arctan\left(\frac{v_y(t)}{v_x(t)}\right) \qquad \mathbf{v}\left(t\right) = v\left(t\right)\begin{pmatrix}\cos(\vartheta)\\ \sin(\vartheta)\end{pmatrix} \tag{5}$$

$$\mathbf{r}\left(0\right) = \begin{pmatrix}0\\ 0\end{pmatrix} \qquad \mathbf{v}\left(0\right) = v_0\begin{pmatrix}\cos(\vartheta_0)\\ \sin(\vartheta_0)\end{pmatrix} \tag{6}$$

In this assignment we are considering gravity as well as drag and spin forces in four different cases:

A. Ideal Object

Without drag $\mathbf{F}_{\text{net}}$ (7) is very straight forward as only gravity acts on the object, and even then only along one axis.

$$\mathbf{F}_{\text{net}} = -mg\ \hat{\mathbf{y}} \tag{7}$$

B. Smooth Object with Drag

A simple approximation of the laminar flow drag force over a smooth surface is $\mathbf{F}_{\text{drag}} = -C\rho A v^2\ \hat{\mathbf{v}}$, where $C = 1/2$, ρ is the density of surrounding fluid, and A is the frontal area of the object. Adding this form of drag to the ideal case results in (8).

$$\mathbf{F}_{\text{net}} = -\frac{1}{2}\rho A v^2 \begin{pmatrix} \cos(\vartheta) \\ \sin(\vartheta) \end{pmatrix} - mg\ \hat{\mathbf{y}} \tag{8}$$

C. Dimpled Object with Drag

We can increase the complexity of the problem by allowing the object's surface to be dimpled, like a golf ball. $\mathbf{F}_{\text{drag}} = -C\rho A v^2\ \hat{\mathbf{v}}$ will still work in this case, but now we let C become a piecewise function of the object's speed (9). This is done to account for the switch between laminar flow at low speeds, where the dimples effects are negligible and $\mathbf{F}_{\text{drag}} \propto v^2$, and turbulent flow at high speeds, where the dimples break up the air flow over the object's surface and allow it to maintain contact longer. The air flow thus sticks close to the surface further along it's length, thereby reducing the overall wake vortex and drag, which can be modeled by letting $\mathbf{F}_{\text{drag}} \propto v$. With this modification our $\mathbf{F}_{\text{net}}$ equation becomes (10).

$$C(v) = \begin{cases} \frac{1}{2} & v \leq v_{\text{transition}} \\ \frac{C'}{v} & v \geq v_{\text{transition}} \end{cases} \qquad \text{where} \qquad C' \leq \frac{v_{\text{transition}}}{2} \tag{9}$$

$$\mathbf{F}_{\text{net}} = -C(v)\,\rho A v^2 \begin{pmatrix} \cos(\vartheta) \\ \sin(\vartheta) \end{pmatrix} - mg\ \hat{\mathbf{y}} \tag{10}$$

D. Dimpled Object with Drag and Spin

Lastly, we can allow the object to spin in addition to the drag force considered earlier. In particular we will be considering objects with backspin, like a golf ball after it leaves the club. The spin will make the relative velocity of the surrounding fluid over the object's surface a function of position, thereby creating uneven drag forces. We can model this effect with a Magnus force, $\mathbf{F}_{\text{Magnus}} = S_0\, \vec{\omega} \times \mathbf{v}$. In our coordinate system backspin corresponds to having $\hat{\omega} = \hat{\mathbf{z}}$ out of the page. With this in mind we can perform the cross product, break $\mathbf{F}_{\text{Magnus}}$ into its components, and compute the new $\mathbf{F}_{\text{net}}$ (11).

$$\mathbf{F}_{\text{net}} = S_0 \omega v \begin{pmatrix} \cos(\vartheta + \frac{\pi}{2}) \\ \sin(\vartheta + \frac{\pi}{2}) \end{pmatrix} - C(v)\,\rho A v^2 \begin{pmatrix} \cos(\vartheta) \\ \sin(\vartheta) \end{pmatrix} - mg\ \hat{\mathbf{y}} \tag{11}$$

It is convenient to define one constant $\eta \equiv \frac{S_0 \omega}{m}$ to hold the S_0 and ω dependence. Making this substitution and simplifying the trigonometry leaves us with the final form of $\mathbf{F}_{\text{net}}$ (12).

$$\mathbf{F}_{\text{net}} = m\eta v \begin{pmatrix} -\sin(\vartheta) \\ \cos(\vartheta) \end{pmatrix} - C(v)\,\rho A v^2 \begin{pmatrix} \cos(\vartheta) \\ \sin(\vartheta) \end{pmatrix} - mg\ \hat{\mathbf{y}} \tag{12}$$

III. RESULTS

We begin by considering the ideal case where only gravity, $g = 9.80665\ \mathrm{m/s^2}$, acts on the golf ball. From introductory mechanics we expect to see perfect parabolas, maximum range at $\vartheta_0 = 45°$, and a symmetry in range for $\vartheta_0 = \vartheta' \pm \vartheta''$, and indeed Figure 1 has these characteristics. While perhaps not as interesting as the cases to come, the ideal case serves as an important validation of our mathematics and code.

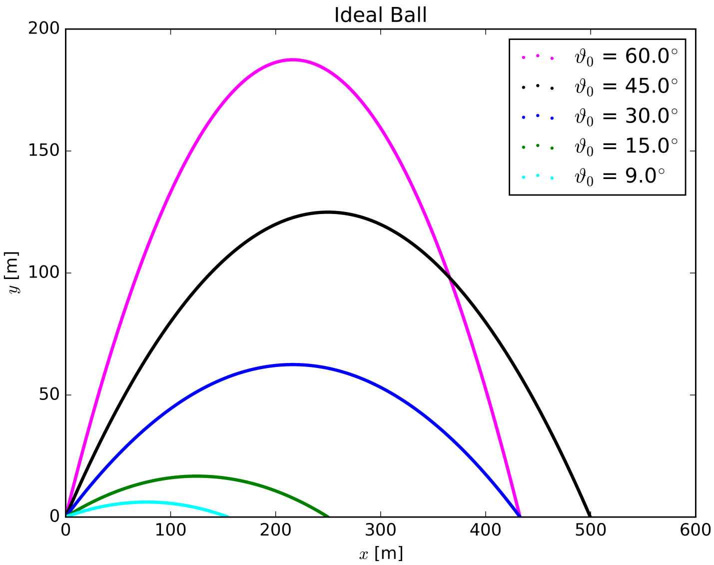

FIG. 1: Trajectories of an ideal golf ball without drag at different initial elevations, ϑ_0. Note that $\vartheta_0 = 45°$ has the maximum range, and that $\vartheta_0 = 30°$ and $60°$ have the same range, as expected.

Next the drag on a smooth golf ball was added, using $\rho_{\text{sea level}} = 1.29\ \mathrm{kg/m^3}$ and $A = 0.0014\ \mathrm{m^2}$. As could be expected, the drag drastically reduces the ranges from the ideal case at all ϑ_0's, less so for flatter trajectories, see Figure 2.

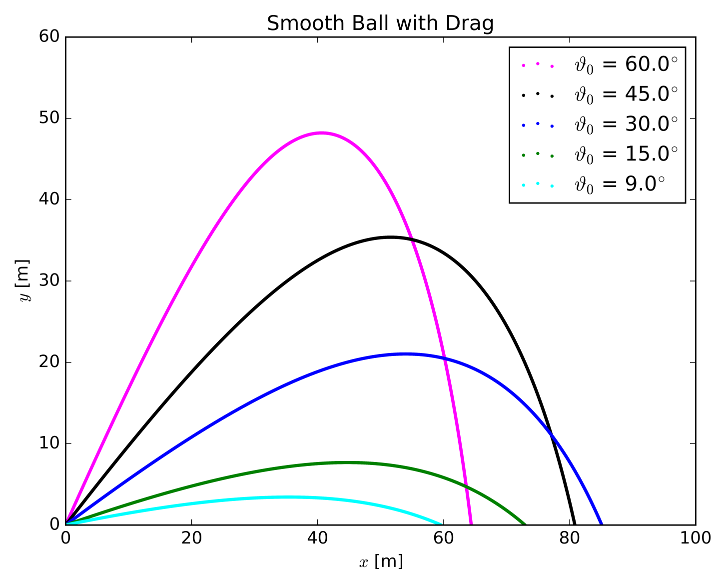

FIG. 2: Trajectories of a smooth golf ball with drag at different initial elevations, ϑ_0. Note the massive decrease in range from the ideal golf ball due to the introduction of drag.

Changing to a dimpled golf ball with $C' = 7$ m/s and $v_{\mathrm{transition}} = 14$ m/s, we find that the reduced drag force at higher velocities, $v \geq v_{\mathrm{transition}}$, gives the ball an increase in range, see Figure 3.

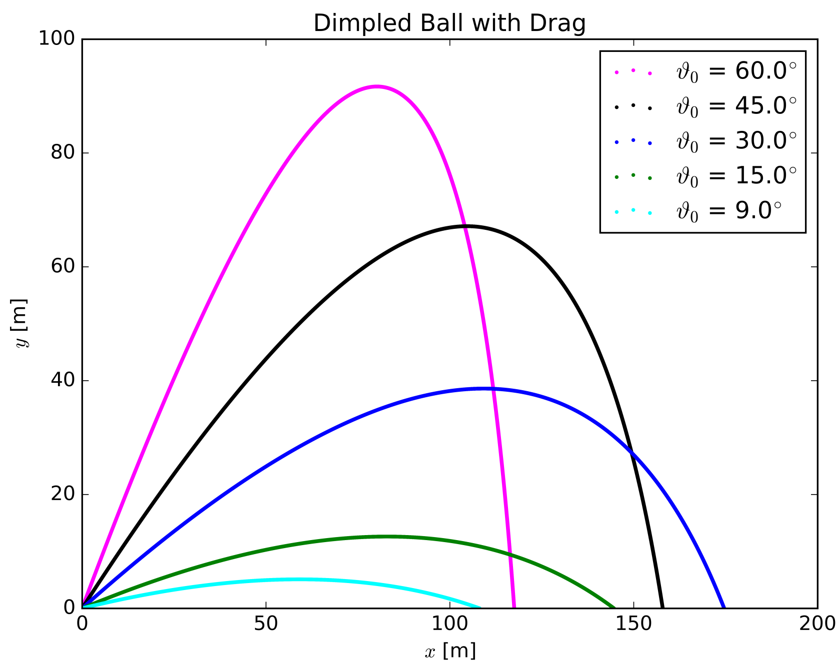

FIG. 3: Trajectories of a dimpled golf ball with drag at different initial elevations, ϑ_0. In comparison to the smooth golf ball it is easy to see that the dimples help the golf ball reduce drag and fly further.

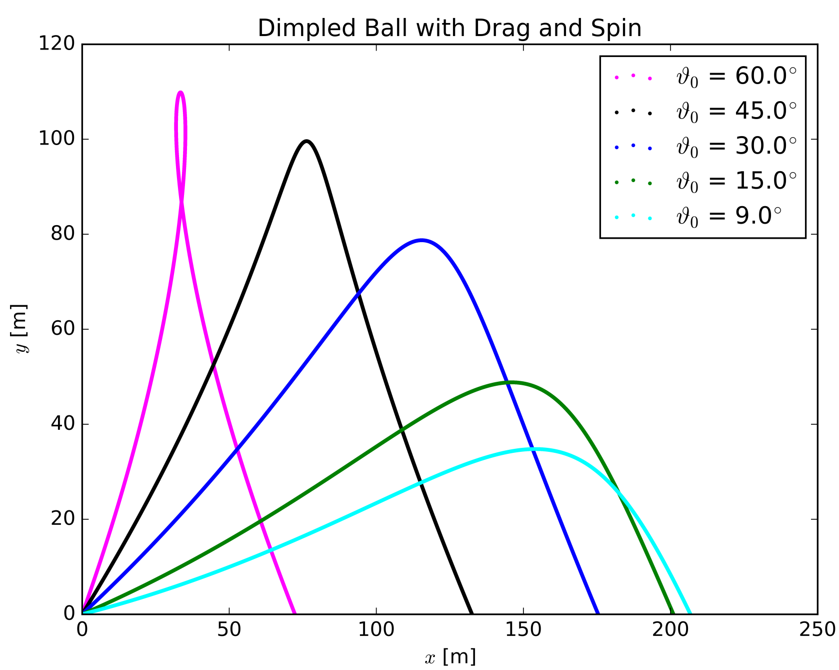

FIG. 4: Trajectories of a dimpled golf ball with drag and spin at different initial elevations, ϑ_0. Note that with backspin the maximum range is now obtained at $\vartheta_0 = 9°$, and that at large enough launch angles, $\vartheta_0 = 60°$, loop–de–loops are possible.

Lastly, the introduction of (back) spin with $\eta = 0.25\ \mathrm{s}^{-1}$ in combination with drag on the dimpled golf ball gives it trajectories familiar to those observed by golfers in the real world, see Figure 4. They are characterized by a relatively straight initial climb over the first half of the flight, before shifting into a more usual parabolic trajectory for the second half. Very shallow launch angles, $\vartheta_0 = 9°$, now have the maximum range as $\mathbf{F}_{\mathrm{Magnus}} \perp \mathbf{v}$ mostly provides the golf ball with extra lift. On the other hand, at very high launch angles, $\vartheta_0 = 60°$, $\mathbf{F}_{\mathrm{Magnus}}$ largely points in $-\hat{\mathbf{x}}$ and can pull the ball through a loop–de–loop. This trajectory in particular would be hard to come by analytically!

IV. CONCLUSIONS

Using the Euler method we were able to computationally determine the trajectories of golf balls in a variety of complex situations that would have been difficult, if not impossible, to analytically derive. The computational method and code was checked against the known behavior of the ideal no drag case to verify it was functioning as intended. The most complex case of a dimpled golf ball undergoing drag with backspin produced trajectories similar to those observed by real world golf balls in flight. Overall, this assignment showcased the power of computational tools in situations where analytical ones fail.

The Python source code used to produce these results can be found online at `http://github.com/mepland/PHYS_566_Computational_HW/tree/master/hw3/code`, and is included in Section VI.

V. SUPPORTING MATERIAL

```

Beginning golf.py simulation
Parameters have units are SI unless otherwise specified
----------------------------------------------

Mass is: 0.046
Initial Velocity is: 70.0
Air Density Rho is: 1.29
Frontal Area A is: 0.0014
g is: 9.80665

Launch Angles Vartheta are:
1.0472 radians, 60.0 degrees
0.7854 radians, 45.0 degrees
0.5236 radians, 30.0 degrees
0.2618 radians, 15.0 degrees
0.1571 radians, 9.0 degrees

Drag Coefficients are C = 0.5 and C' = 7.0
Transition Velocity is: 14.0

Spin Constant Eta is: 0.25

Simulation Time Step is: 0.00050
Simulation Max Step Size is: 0.050
----------------------------------------------
----------------------------------------------


Running Case: Ideal Ball
----------------------------------------------

Running with vartheta0 = 60.0 degrees
Running with vartheta0 = 45.0 degrees
Running with vartheta0 = 30.0 degrees
Running with vartheta0 = 15.0 degrees
Running with vartheta0 = 9.0 degrees
Case Completed

Running Case: Smooth Ball with Drag
----------------------------------------------

Running with vartheta0 = 60.0 degrees
Running with vartheta0 = 45.0 degrees
Running with vartheta0 = 30.0 degrees
Running with vartheta0 = 15.0 degrees
Running with vartheta0 = 9.0 degrees
Case Completed

Running Case: Dimpled Ball with Drag
----------------------------------------------

Running with vartheta0 = 60.0 degrees
Running with vartheta0 = 45.0 degrees
Running with vartheta0 = 30.0 degrees
Running with vartheta0 = 15.0 degrees
Running with vartheta0 = 9.0 degrees
Case Completed

Running Case: Dimpled Ball with Drag and Spin
----------------------------------------------

Running with vartheta0 = 60.0 degrees
Running with vartheta0 = 45.0 degrees
Running with vartheta0 = 30.0 degrees
Running with vartheta0 = 15.0 degrees
Running with vartheta0 = 9.0 degrees
Case Completed


Done!
```

output.log

PHY260 Homework #3

Due Date: Feb. 11th, 5:00pm via Sakai

Golf

Write a program to calculate the trajectory of a golf ball of mass 46 grams and calculate the trajectories as a function of angle (use $\vartheta = 45^\circ, 30^\circ, 15^\circ$ and 9°). Choose the initial velocity of the golf ball to be 70 m/s. For the drag, assume a general form of

$$F_{\text{drag}} = -C\,\rho\,A\,v^2 \tag{1}$$

where ρ is the density of air (at sea level), 1.29 kg/m^3, A is the frontal area of the golf ball, 0.0014 m^2, and C is a coefficient to be discussed below. For each angle, calculate and compare the trajectories for the following cases:

a) ideal trajectory: no drag and no spin [2 points]

b) smooth golf ball with drag: choose $C = 1/2$ [2 points]

c) dimpled golf ball with drag: choose $C = 1/2$ for speeds up to $v = 14$ m/s and $C = 7.0/v$ at higher velocities. This transition to a reduced drag coefficient is due to turbulent flow, caused by the dimples. [3 points]

d) dimpled golf ball with drag and spin: use a Magnus force $\vec{F} = S_0\,\vec{\omega} \times \vec{v}$ with a backspin of $S_0\omega/m = 0.25s^{-1}$ for a typical case. [3 points]

Please note that from now on source code lacking comments or unsafe programming (e.g. risking division by zero or over/underflow of arrays and variables) will lead to a 1 point deduction, respectively.

Your homework submission should consist of:

- a document outlining the problem, detailing your solution and discussion of your results - the document should include the requested figures. The document should be in pdf format and you should use colors and different marker symbols to enhance the readability of your figures.
- the source code of your program

Both files should be submitted as separate attachments on Sakai

VI. CODE

```
#!/usr/bin/env /home/mbe9/Documents/Spring_2016/PHYS_566_Computational_HW/anaconda/bin/python
# #! to the correct python install, not portable!!!

# Run by calling
# ./golf.py 2>&1 | tee output.log
# to save output to output.log

import os
import sys
import numpy as np
import matplotlib.pyplot as plt

########################################################
# Set fixed parameters, SI Units
mass = 46*0.001 # mass of ball = 46 grams
v0 = 70.0 # Initial velocity
rho = 1.29 # Density of air (at sea level)
A =  0.0014 # Frontal Area
g = 9.80665 # g at sea level

vartheta0 = [60, 45, 30, 15, 9] # Initial launch angles (degrees)
# Convert to radians
for k in range(len(vartheta0)):
  vartheta0[k] = vartheta0[k]*(np.pi/180.0)

# Drag Coefficients and transition velocity
C = 0.5
C_prime = 7.0
v_transition = 14.0

# Spin Constant eta defined in writeup
eta = 0.25

dt = 0.0005 # Time step of simulation, empirically set to satisfy max_r
max_r = 0.05 # Maximum distance allowed between time steps
# For comparison, Solving 0.0014 = 2 pi r^2 for half the surface area of a sphere gives us the balls radius r ~= 0.015


########################################################
# Print out starting values
print '\nBeginning golf.py simulation'
print 'Parameters have units are SI unless otherwise specified'
print '---------------------------------------------'

print '\nMass is: %.3f' % mass
print 'Initial Velocity is: %.1f' % v0
print 'Air Density Rho is: %.2f' % rho
print 'Frontal Area A is: %.4f' % A
print 'g is: %.5f' % g

print '\nLaunch Angles Vartheta are: '
for k in range(len(vartheta0)):
  print '%.4f radians, %.1f degrees' % (vartheta0[k], vartheta0[k]*(180.0/np.pi))

print '\nDrag Coefficients are C = %.1f and C\' = %.1f' % (C, C_prime)
print 'Transition Velocity is: %.1f' % v_transition

print '\nSpin Constant Eta is: %.2f' % eta

print '\nSimulation Time Step is: %.5f' % dt
print 'Simulation Max Step Size is: %.3f' % max_r
print '---------------------------------------------'
print '---------------------------------------------\n'

########################################################
# Define time_step class to hold all the relevant parameters at a time step
class time_step:

    def __init__(self, time):
  self.t = time
        self.x = -99.0
  self.y = -99.0
  self.vx = -99.0
  self.vy = -99.0

  # Return vartheta of the time step when called
    def vartheta(self):
  if self.x == -99.0 and self.y == -99.0 and self.vx == -99.0 and self.vy == -99.0: print 'WARNING COMPUTING THETA ON DEFAULT TIME STEP'
```

```
  return np.arctan2(self.vy, self.vx)

  # Return the speed v of the time step when called
    def vmag(self):
  if self.x == -99.0 and self.y == -99.0 and self.vx == -99.0 and self.vy == -99.0: print 'WARNING COMPUTING VMAG ON DEFAULT TIME STEP'
  return np.sqrt(np.square(self.vx) + np.square(self.vy))

  # Return spatial separation r from another time step
    def r(self, time_step2):
  if self.x == -99.0 and self.y == -99.0 and self.vx == -99.0 and self.vy == -99.0: print 'WARNING COMPUTING r WITH ON DEFAULT TIME STEP'
  return np.sqrt( np.square(self.x - time_step2.x) + np.square(self.y - time_step2.y))
    # Note: We could speed up the computation by saving r, vmag after it is called the first time
    # then return the value we have. This will be more complex and cost more memory though
    # The plot printing seems to be the bottleneck, so it's not worth changing

# end class for time_step

########################################################
# Define a function to run the simulation based on which case we are in and what vartheta0 we want
def run_sim(case, m_vartheta0, m_color):

  # Start the list of time_steps
  run = [time_step(0.0)]

  # Set the initial values
  run[0].x = 0.0
  run[0].y = 0.0
  run[0].vx = v0*np.cos(m_vartheta0)
  run[0].vy = v0*np.sin(m_vartheta0)

  # Loop until we hit the ground
  current_y = 99.0 # use to halt while loop
  i = 0 # current time step
  C_dimpled = C # Declare C_dimpled for use later, just set it to C for now

  while current_y > 0.0:
    i = i + 1 # increment time step
    run.append(time_step(i*dt)) # append a new time_step object

    # Compute the new position
    run[i].x = run[i-1].x + run[i-1].vx*dt
    run[i].y = run[i-1].y + run[i-1].vy*dt

    # Compute the new velocity, see writeup for physics...

    if(case == 'a'): # ideal
      run[i].vx = run[i-1].vx + (dt/mass)*( 0.0 )
      run[i].vy = run[i-1].vy + (dt/mass)*( -mass*g )

    if(case == 'b'): # smooth ball with drag
      F_drag_smooth = -C*rho*A*np.square(run[i-1].vmag())
      run[i].vx = run[i-1].vx + (dt/mass)*( F_drag_smooth*np.cos(run[i-1].vartheta()) )
      run[i].vy = run[i-1].vy + (dt/mass)*( F_drag_smooth*np.sin(run[i-1].vartheta()) -mass*g )

    if(case == 'c'): # dimpled ball with drag
      # see what v regime we are in and pick C accordingly
      if( run[i-1].vmag() <= v_transition):
        C_dimpled = C # 1/2
      else:
        C_dimpled = C_prime/run[i-1].vmag() # C'/v
      F_drag_dimpled = -C_dimpled*rho*A*np.square(run[i-1].vmag())
      run[i].vx = run[i-1].vx + (dt/mass)*( F_drag_dimpled*np.cos(run[i-1].vartheta()) )
      run[i].vy = run[i-1].vy + (dt/mass)*( F_drag_dimpled*np.sin(run[i-1].vartheta()) -mass*g )

    if(case == 'd'): # dimpled ball with drag and spin
      # see what v regime we are in and pick C accordingly
      if( run[i-1].vmag() <= v_transition):
        C_dimpled = C # 1/2
      else:
        C_dimpled = C_prime/run[i-1].vmag() # C'/v
      F_drag_dimpled = -C_dimpled*rho*A*np.square(run[i-1].vmag())
      F_spin = mass*eta*run[i-1].vmag()
      run[i].vx = run[i-1].vx + (dt/mass)*( -F_spin*np.sin(run[i-1].vartheta()) + F_drag_dimpled*np.cos(run[i-1].vartheta()) )
      run[i].vy = run[i-1].vy + (dt/mass)*(  F_spin*np.cos(run[i-1].vartheta()) + F_drag_dimpled*np.sin(run[i-1].vartheta()) -mass*g )


    # Make sure we didn't move to far, notify user and exit if we did
    if(run[i].r(run[i-1]) > max_r):
      print 'On time step %d the ball moved to far, r = %.3f\nProgram Exiting' % (i, run[i].r(run[i-1]))
```

```
      sys.exit()
    current_y = run[i].y # update current_y
  # end while current_y > 0.0 loop

  ##########################################################
  # Create ndarrays that we can plot

  # Declare the x and y ndarrays based on our finished run's size
  x_ndarray = np.zeros(len(run))
  y_ndarray = np.zeros(len(run))

  # Fill the ndarrays
  for j in range(len(run)):
    x_ndarray[j] = run[j].x
    y_ndarray[j] = run[j].y

  ##########################################################
  # Create and return the scatter object
  m_label = '$\\vartheta_{0}$ = %.1f$^{\circ}$' % (m_vartheta0*(180.0/np.pi))
  return plt.scatter(x_ndarray, y_ndarray, marker='.', label=m_label, c=m_color, edgecolors='none')
# end def for run_sim

##########################################################
# Define a function to run the simulation multiple times for all the vartheta0's
def loop_vartheta0(case):
  # Define the colors we want to use, include some extra to be safe
  # If crimson starts printing, you better be careful!
  colors = ['magenta', 'black', 'blue', 'green', 'cyan', 'crimson']
  scatters = []
  for k in range(len(vartheta0)):
    print 'Running with vartheta0 = %.1f degrees' % (vartheta0[k]*(180.0/np.pi))
    scatters.append(run_sim(case, vartheta0[k], colors[k]))
  return scatters
# end def for loop_vartheta0

##########################################################
# Define a function to run, plot, and print a whole case
def run_case(case, title, fname):
  print '\nRunning Case: %s' % title
  print '---------------------------------------------\n'

  fig = plt.figure(case) # get a separate figure for the case
  ax = fig.add_subplot(111) # Get the axes, effectively

  loop_vartheta0(case) # generate the scatters

  ###########################
  # Format the plot

  ax.set_title(title)

  ax.set_xlabel('$x$ [m]')
  ax.set_ylabel('$y$ [m]')

  ax.set_xlim((0.0, None))
  ax.set_ylim((0.0, None))

  plt.legend()

  ###########################
  # Print the plot
  # fig.savefig(output_path+'/'+fname+'.pdf')
  # Due to the number of points pdfs will be VERY large
  # So instead only make high quality pngs
  fig.savefig(output_path+'/'+fname+'.png', dpi=900)

  print 'Case Completed'
# end def for run_case



##########################################################
# Create the output dir, if it already exists don't crash, otherwise raise an exception
# Adapted from A-B-B's response to http://stackoverflow.com/questions/273192/in-python-check-if-a-directory-exists-and-create-it-if-necessary
# Note in python 3.4+ 'os.makedirs(output_path, exist_ok=True)' would handle all of this...
output_path = './output' # Set output path, hard coded...
try:
    os.makedirs(output_path)
except OSError:
    if not os.path.isdir(output_path):
        raise Exception('Problem creating output dir %s !!!\nA file with the same name probably already exists, please fix the conflict and run again.' % output_path)
```

```

#########################################################
# Finally, actually run over the four cases

run_case('a', 'Ideal Ball', 'ideal')
run_case('b', 'Smooth Ball with Drag', 'smooth_ball_w_drag')
run_case('c', 'Dimpled Ball with Drag', 'dimpled_ball_w_drag')
run_case('d', 'Dimpled Ball with Drag and Spin', 'dimpled_ball_w_drag_and_spin')


#########################################################
print '\n\nDone!\n'
```

golf.py