PHYC 2050 Class Notes

Thomas J. Duck

Department of Physics and Atmospheric Science
Dalhousie University

tom.duck@dal.ca

March 5, 2010

CHAPTER

FOUR

PROJECTILE MOTION: GOLF

... otherwise known as Chapter Fore. ;^)

In this chapter we consider the flight of a golf ball, built up in steps. We first look at projectile motion in a vacuum; i.e., the standard projectile motion problem. We then add molecular friction from air, and follow with spin (and dimples!). Although the problem starts off as one that we can treat analytically (pencil and paper math), it ends such that our only recourse is a numerical solution (on the computer). In each case we write down Newton's Second Law and proceed to write finite difference equations.

Here are a few things you need to know about golf. The mass of a USGA-approved golf ball can be at most 45.93 grams, and the diameter can be no smaller than 42.67 mm (source: Wikipedia). A good player will drive the ball at something like 70 m/s, with a launch angle of about 15 deg. Add another ten to twenty meters per second to the drive speed if you are Tiger Woods. The spin rate varies from golfer to golfer, and depends on the club being used and the mechanics of the golfer's stroke. Typical values range from 2000 to 4000 rpm for a drive, and higher for the irons. Golf balls are dimpled, and this has a profound effect on its trajectory, as we shall see.

4.1 Vertical Motion in a Vacuum

Consider a spherical projectile of mass m in moving vertically in a vacuum. The free-body diagram is very simple (see Figure 4.1). There is only one force acting on the sphere: gravity, a vector quantity that always point downward.

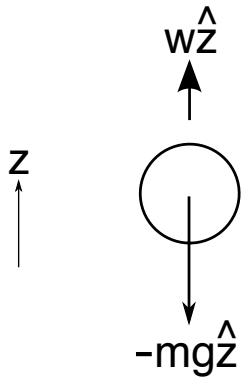

Figure 4.1: Free body diagram for vertical motion in a vacuum.

Newton's Second Law states

$$\sum \vec{F} = m\vec{a}$$

i.e., the sum of all forces equals the mass times acceleration. For the case of purely vertical motion in a vacuum, the equation of motion is therefore

$$-mg = m\frac{d^2z}{dt^2}$$

where z is the altitude,

$$w \equiv \frac{dz}{dt}$$

is the vertical speed and

$$\frac{dw}{dt} = \frac{d^2z}{dt^2}$$

is the vertical acceleration. Dividing out the mass of the object leaves

$$\frac{d^2z}{dt^2} = -g$$

as the equation of vertical motion.

Suppose that the initial vertical speed is w_0, and the initial vertical altitude is z_0. The analytical solution to the equation of motion with these initial conditions is

$$z = z_0 + w_0 t - \frac{1}{2}gt^2$$

Proof:

$$\begin{aligned} RHS &= \frac{d^2z}{dt^2} \\ &= \frac{d}{dt}\frac{dz}{dt} \\ &= \frac{d}{dt}\left[\frac{d}{dt}\left(z_0 + w_0 t - \frac{1}{2}gt^2\right)\right] \\ &= \frac{d}{dt}\left(w_0 - gt\right) \\ &= -g \\ &= LHS \end{aligned}$$

To solve the equation numerically, we begin by marking each non-constant term with time index i as follows:

$$\left(\frac{d^2z}{dt^2}\right)_i = -g$$

Substituting the finite difference formula for second derivatives gives

$$\frac{z_{i+1} - 2z_i + z_{i-1}}{h^2} = -g$$

and rearranging yields

$$z_{i+1} = -gh^2 + 2z_i - z_{i-1}$$

where h is the time step. The equation indicates that the altitude at the *next* time step can be determined from the altitudes at the *current* and *previous* time steps. To begin we will need two points, z_0 and z_1. The above equation can then be used to calculate z_2, and thereafter z_3 (from z_1 and z_2), and so on.

Suppose we know the launch altitude z_0 and and speed w_0. How can we use this information to calculate z_1? Recall that the vertical speed is given by

$$w \equiv \frac{dz}{dt}$$

Evalulating at time index 0 gives

$$w_0 = \left(\frac{dz}{dt}\right)_0$$

Using the forward difference formula for the derivative gives

$$w_0 = \frac{z_1 - z_0}{h}$$

and rearranging yields

$$z_1 = w_0 h + z_0$$

So, z_1 is determined by its own difference formula. Although it doesn't take the acceleration due to gravity into account, we assume that for small enough time step h this will only result in a small error.

Here is where we run into a programming problem. We want to track the motion of the projectile while it is above ground. Unfortunately, we don't know how many time steps it will take to land, and so we can't determine the output array lengths, or how many times to iterate in a for loop a priori.

The best approach in this case is to store the z_i values in a list, and append each new altitude we calculate until the projectile hits ground. But where do we find functions that can do useful things with a list, like append a value?

Functions are normally contained in modules, but there is no list module given in the python standard library. It turns out that functions and variables associated with a specific type can be attached directly to an object, and that is what is done for lists. The functions and variables are referred to as *methods* and *attributes* in this context. So, for example, a list contains methods that can be used for sorting or to append an item. Numpy arrays have a method for summation and an attribute which describes the data type of its numbers. You can look at any object's methods and attributes by using the python `help()` function or reading the online documentation.

Here is a simple example of how the `append()` method of a list is invoked:

```
>>> z = [0,1]
>>> z.append(2)
>>> print z
[0, 1, 2]
```

You can see that the notation is a lot like what we used for modules before.

Continuing, the implementation of the finite difference solution, checked against the analytical "truth" is:

```
#! /usr/bin/env python

# golf_vacuum_1d.py

import numpy, pylab
import matplotlib

h = 0.01                    # Time step (s)
g = 9.8                     # Gravity (m/s^2)

z0 = 0.0                    # Initial altitude (m)

V0 = 70.0                   # Launch speed (m/s)
theta0 = numpy.radians(15)  # Launch angle (radians)

w0 = V0*numpy.sin(theta0)   # Initial vertical speed (m/s)

# Create lists to store the trajectory
z = [z0, w0*h + z0]  # Altitudes (m)
t = [0, h]           # Times (s)

# Perform the finite difference calculation
i = 2
while z[-1] > 0:
    tnext = i*h
    znext = -g*h**2 + 2*z[-1] - z[-2]

    t.append(tnext)
    z.append(znext)

    i = i+1  # Increment counter

# Convert lists to arrays
t = numpy.array(t)
z = numpy.array(z)

# Determine the absolute error
z2 = z0 + w0*t - 0.5*g*t**2  # Truth
err = z-z2

# Plotting
pylab.figure(1)
pylab.plot(t,z,label='Numerical')
pylab.plot(t,z2,label='Analytical')
pylab.xlabel('t (s)')
pylab.ylabel('z (m)')
pylab.ylim(0.,25.)

font = matplotlib.font_manager.FontProperties(size='small')
legend = pylab.legend(loc='upper left',prop=font)
legend.draw_frame(False)

pylab.subplots_adjust(left=0.3,right=0.7,top=0.7,bottom=0.3)

pylab.figure(2)
pylab.plot(t,err)
pylab.xlabel('t (s)')
```

```
pylab.ylabel('Error (m)')
pylab.subplots_adjust(left=0.3,right=0.7,top=0.7,bottom=0.3)

pylab.show()
```

Plots of altitude versus time (Figure 4.2) and error versus time (Figure 4.3) are given below. The golf ball flies for about 4 seconds, which is somewhat shorter than expectations. Notice that the overall error is quite small but increases with time. The net error can be reduced by reducing the time step `h`, but only up to the limit of machine precision.

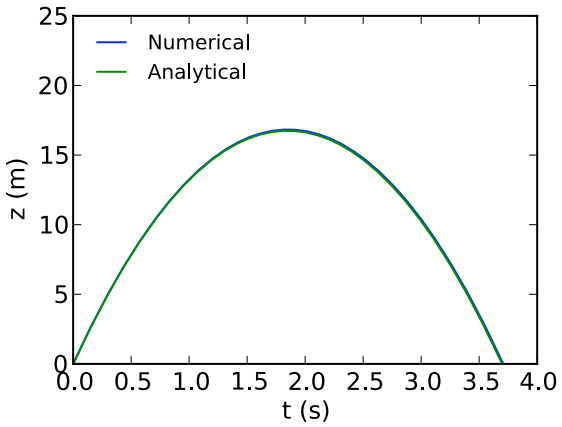

Figure 4.2: Altitude versus time for the numerical and analytical solutions.

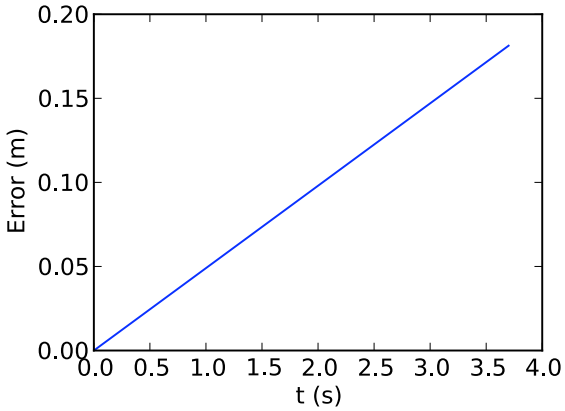

Figure 4.3: Error in the numerical solution.

4.2 Two-dimensional Motion in a Vacuum

The free-body diagram may be adapted for 2-dimensional motion (see Figure 4.4).

The horizontal and vertical distances are x and z, and the wind speed components in those directions are u and w, respectively. The total velocity vector $\vec{V}$ makes angle θ with the horizontal. Thus,

$$u = V \cos\theta$$

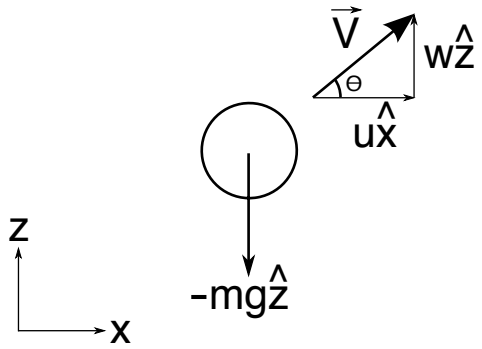

Figure 4.4: Free-body diagram for two-dimensional motion in a vacuum.

and

$$w = V \sin\theta$$

where

$$V \equiv |\vec{V}|$$

These equations can be used to determine the initial speed components $u0$ and *w0*, assuming that the launch speed $V0$ is provided.

Equations of motion can be written by applying Newton's Second Law in each of the cardinal directions. There is still only one force acting on the sphere: gravity, which acts in the vertical direction only. Thus, the equation of motion in the vertical direction remains the same as before. The equation of motion in the horizontal (after the mass has been divided out) is

$$\frac{d^2x}{dt^2} = 0$$

where x is the horizontal distance,

$$u \equiv \frac{dx}{dt}$$

is the horizontal speed, and

$$\frac{du}{dt} = \frac{d^2x}{dt^2}$$

is the horizontal acceleration.

Suppose the the initial horizontal position is x_0 and the speed is u_0. The general solution to the horizontal equation of motion is

$$x = x_0 + u_0 t$$

Rearranging for t and substituting into the vertical component solution gives

$$z = z_0 + \frac{w_0}{u_0}(x - x_0) - \frac{g}{2u_0^2}(x - x_0)^2$$

which is the equation for a parabola; i.e., the projectile will follow a parabolic trajectory.

The finite difference form of the equation of motion is

$$\frac{x_{i+1} - 2x_i + x_{i-1}}{h^2} = 0$$

Rearranging yields

$$x_{i+1} = 2x_i - x_{i-1}$$

Two previous points are needed to calculate the next position on the x-axis. We can use the same trick from before:

$$u_0 = \left(\frac{dx}{dt}\right)_0$$

$$\rightarrow u_0 = \frac{x_1 - x_0}{h}$$

$$\rightarrow x_1 = u_0 h + x_0$$

The equations can be coded into the previous program, with some simple modifications to produce the following plot:

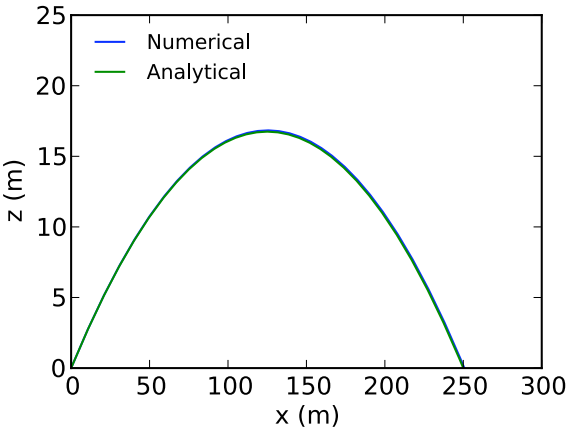

Figure 4.5: Altitude versus horizontal distance for the numerical and analytical solutions.

The golf ball travels about 250 m downrange. This seems reasonable for a good golfer, although we haven't yet included the effect of air resistance in our simulation. The numerical and analytical solutions continue to agree quite well.

4.3 Air Resistance and Drag

We now consider two-dimensional projectile motion through a fluid so that there is air resistance. Analytical solutions are still possible here for simple cases (e.g., Timmerman and van der Weele, Am. J. Phys., 1998), but they are very complicated and so we won't consider them further.

The magnitude of the drag force is given by

$$F_D = \frac{1}{2} C_D \rho A V^2$$

where C_D is the drag coefficient, ρ is the mass density of air ($1.2\ kg/m^3$ at sea level), A is the cross-sectional area of the projectile, and

$$V = (u^2 + w^2)^{1/2}$$

is its speed. The equation for drag makes intuitive sense: for example, a rider on a bicycle experiences greater air resistance when traveling faster, sitting straight up (greater A), and in thicker air (greater rho). C_D is an experimentally-determined "fudge factor" that calibrates the drag force to the proper strength. It depends on the object's shape and other factors such as speed and spin rate (i.e., it is not necessarily a constant).

The drag coefficient for a smooth sphere is $C_D = 0.5$ so long as the speed is not too great. However, golf balls have dimples, and they have an amazing effect. For speeds greater than a critical value of about $V_c = 14\ m/s$ the drag coefficient drops roughly as $1/V$ from a maximum of 0.5 to a lower limit of about 0.25 (Mehta, Ann. Rev. Fluid Mech., 1985).

Why should the drag decrease for the dimpled ball as the speed increases? It turns out that the dimples act to drag a little air along with the ball, and this allows it to essentially slip through the atmosphere with lower friction. In fluid dynamics terms, the dimples induce a turbulent boundary layer that allow the flow streamlines to separate from the surface of the ball further aft, resulting in a narrower wake and lower drag (see http://www.aerospaceweb.org/question/aerodynamics/q0215.shtml for some diagrams and further explanation). Enough about fluid dynamics! Although it is a subject I love, the point we are interested in right now is that dimples reduce the drag and so allow a golf ball to travel further than a smooth sphere.

The drag force is directed opposite to the motion of the projectile, as shown in the free-body diagram of Figure 4.6.

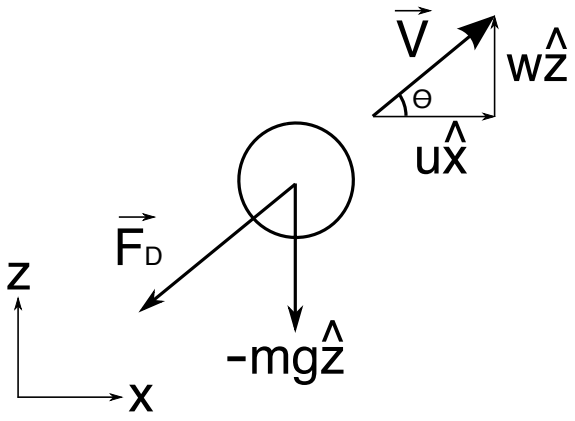

Figure 4.6: Free-body diagram for the projectile with drag.

The drag components F_{Dx} and F_{Dz} are needed in the x- and z-directions, respectively. They are

$$\begin{aligned} F_{Dx} &= -\frac{1}{2}C_D\rho AV^2\cos\theta \\ &= -\frac{1}{2}C_D\rho AV^2\frac{u}{V} \\ &= -\frac{1}{2}C_D\rho AVu \end{aligned}$$

and similarly

$$F_{Dz} = -\frac{1}{2}C_D\rho AVw$$

The equations of motion are therefore

$$\frac{d^2x}{dt^2} = -\frac{\rho AV}{2m}C_Du$$

and

$$\frac{d^2z}{dt^2} = -g - \frac{\rho AV}{2m}C_Dw$$

where the parameters have been rearranged into an order that will prove useful later.

Note that the terminal speed w_T is attained for purely vertical motion with no acceleration:

$$w_T = \left(\frac{2mg}{C_DA\rho}\right)^{1/2}$$

The corresponding finite difference equations are

$$x_{i+1} = -\frac{\rho A}{2m}C_DV_iu_ih^2 + 2x_i - x_{i-1}$$

and

$$z_{i+1} = -(g + \frac{\rho A}{2m}C_DV_iw_i)h^2 + 2z_i - z_{i-1}$$

where

$$V_i = \sqrt{u_i^2 + v_i^2} \quad .$$

Note that C_D changes along the trajectory, and so we must re-calculate it as the projectile's speed changes. For the sake of simplicity, we take $C_D = 0.5$ for speeds lower than 14 m/s, and $C_D = 0.5V_c/V$ for greater speeds, with a lower limit of $C_D = 0.25$. More realistic results require a more careful parameterization based on measurements, and we will introduce one in the next section.

The speed components u_i and w_i are obtained using the backward difference formulae

$$u_i = \frac{x_i - x_{i-1}}{h}$$

and

$$w_i = \frac{z_i - z_{i-1}}{h}$$

Below is the source code. Calculations are performed for both the smooth sphere and the dimpled golf ball, and compared against the previous case of motion in a vacuum.

```
#! /usr/bin/env python

# golf_drag.py

import numpy, pylab
import matplotlib


h = 0.01                        # Time step (s)
g = 9.8                         # Gravity (m/s^2)
dens = 1.2                      # Air density (kg/m^3)

m = 45.93e-3                    # Mass of golf ball (kg)
d = 42.67e-3                    # Ball diameter (m)
A = numpy.pi*(d/2.)**2          # Cross-sectional area of golf ball (m^2)

x0 = 0.0                        # Initial horizontal distance (m)
z0 = 0.0                        # Initial altitude (m)

V0 = 70.0                       # Launch speed (m/s)
theta0 = numpy.radians(15)      # Launch angle (radians)

u0 = V0 * numpy.cos(theta0)     # Initial horizontal speed (m/s)
w0 = V0 * numpy.sin(theta0)     # Initial vertical speed (m/s)


# Perform the finite difference calculation
def get_trajectory(Vc):

    # Create lists to store the trajectory
    x = [x0, u0*h + x0]  # Horizontal distances (m)
    z = [z0, w0*h + z0]  # Altitudes (m)

    while z[-1] > 0:

        # Get the speeds
        u = (x[-1]-x[-2]) / h
        w = (z[-1]-z[-2]) / h
        V = numpy.sqrt(u**2 + w**2)

        # Determine the drag coefficient
        if V<Vc:
            Cd = 0.5
        else:
```

```
            Cd = max(0.5*Vc/V,0.25)

        # Determine the next point in the trajectory
        xnext = -dens*A*V/(2*m)*Cd*u*h**2 + 2*x[-1] - x[-2]
        znext = -(g+dens*A*V/(2*m)*Cd*w)*h**2 + 2*z[-1] - z[-2]

        # Store the values
        x.append(xnext)
        z.append(znext)

    # Convert lists to arrays and return
    return numpy.array(x), numpy.array(z)


# Get trajectories for different Vc values
x1,z1 = get_trajectory(100.)    # Smooth ball
x2,z2 = get_trajectory(14.)     # Golf ball

# Get the trajectory without drag for comparison
x3 = numpy.arange(0,300.)
z3 = z0 + (w0/u0)*(x3-x0) - g/(2*u0**2)*(x3-x0)**2

# Select above-ground points only
x3 = x3.compress(z3>=0.)
z3 = z3.compress(z3>=0.)


# Plotting

pylab.plot(x1,z1,label='Smooth sphere')
pylab.plot(x2,z2,label='Golf ball')
pylab.plot(x3,z3,label='Vacuum')

pylab.xlabel('x')
pylab.ylabel('z')

font = matplotlib.font_manager.FontProperties(size='small')
legend = pylab.legend(loc='upper left',prop=font)
legend.draw_frame(False)

pylab.ylim(0.,25.)

pylab.subplots_adjust(left=0.3,right=0.7,top=0.7,bottom=0.3)

pylab.show()
```

Notice in the program that much of the calculation has been written into a function called `get_trajectory()` so that we can determine trajectories for different parameter choices without resorting to duplicating blocks of code. Our calculations will be *encapsulated* in this way, but perhaps with changed argument lists, in all of the programs that follow.

We have dropped the counter `i`. It is no longer needed: we will not be worrying about the time of flight, and nothing in our implementation needs to be indexed.

Figure 4.7 shows that drag dramatically reduces the distance a projectile can fly. The range is now well short of our expectations. Note also that a dimpled ball travels considerably further than a smooth sphere due to the decreased drag coefficient.

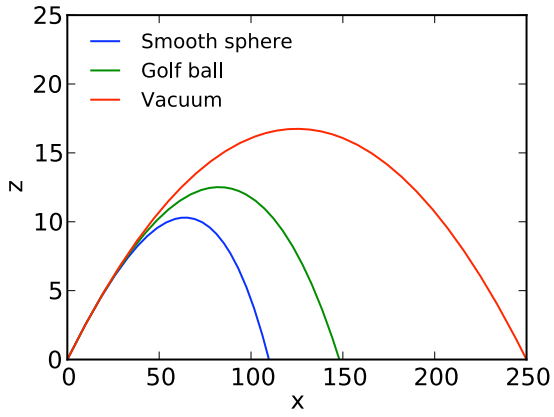

Figure 4.7: Trajectories for a smooth sphere and a dimpled golf ball traveling through the atmosphere, and through a vacuum for comparison.

4.4 Spin and the Magnus Force

In golf, backspin is necessary to hit the ball the maximum distance possible. We can model the backspin of a projectile by including the so-called Magnus force (see Figure 4.8), which provides a *lift* perpendicular to the direction of motion.

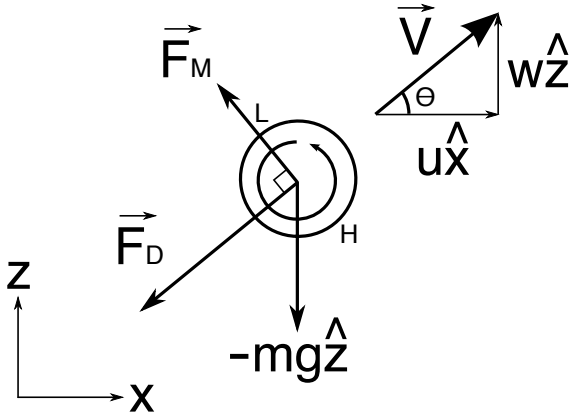

Figure 4.8: Free-body diagram for the projectile with drag and the Magnus force.

The Magnus force results from the fact that the bottom surface of the ball is moving faster relative to air than the upper surface. The relatively high viscous drag on the bottom causes a high-pressure anomaly, and the relatively low viscous drag on the top causes a low-pressure anomaly, both consistent with the Bernoulli effect. The pressure difference between the top and the bottom is the source of the lift.

The Magnus force is given by

$$\vec{F}_M = S_0 \vec{\omega} \times \vec{V}$$

for spin vector $\vec{\omega}$ and S_0 is a lifting parameter that depends on the characteristics of the ball and its flight (i.e., another experimentally-determined "fudge factor"). The magnitude of the spin vector is the angular frequency (measured in

radians/s), and its direction is given by the right-hand rule. The equation is a result from fluid dynamics for the so-called "dynamic pressure" force. The equation should make intuitive sense: the lift is greater when the ball rotates quickly and travels fast. Both the non-spinning ball and the one spinning in place experience no lift.

If the spin vector is perpendicular to the x-y plane then the magnitude of the Magnus force is

$$F_M = S_0 \omega V \quad .$$

The magnitude of the Magnus force can be written similar to the drag force as

$$F_M = \frac{1}{2} C_L \rho A V^2$$

where C_L is the lift coefficient. Usually C_L is determined in experiments rather than S_0. Comparing the two equations for F_M greveals

$$C_L = \frac{2S_0}{\rho A} \frac{\omega}{V} \quad .$$

Thus, we expect the lift coefficient to depend on the parameter ω/V.

Experiments show that a good parameterization for the lift coefficient for golf balls is

$$C_L = 0.3996 + 0.1583 \ln(R) + 0.03790 R^{-0.5}$$

(source: USGA "Second Report on the Study of Spin Generation", 2007, Appendix C).

Here, the spin ratio

$$R = \frac{\omega r}{V}$$

is the ratio of the equatorial speed of the ball to it's airspeed (r is the golf ball's radius). The accompanying parameterization for the drag coefficient is

$$C_D = 0.1403 - 0.3406 R \ln(R) + 0.3747 R^{1.5}$$

These parameterizations are valid for spin ratios between about 0.05 and 2.

As with the drag, we must calculate the lift components in the x and z directions using trigonometry. They are

$$\begin{aligned} F_{Mx} &= -\frac{1}{2} C_L \rho A V^2 \sin\theta \\ &= -\frac{1}{2} C_L \rho A V^2 \frac{w}{V} \\ &= -\frac{1}{2} C_L \rho A V w \end{aligned}$$

and similarly

$$F_{Mz} = +\frac{1}{2} C_L \rho A V u \quad .$$

The equations of motion are therefore

$$\frac{d^2x}{dt^2} = -\frac{\rho A V}{2m}(C_D u + C_L w)$$

and

$$\frac{d^2z}{dt^2} = -g - \frac{\rho A V}{2m}(C_D w - C_L u)$$

Expressing the equations in finite difference form leads to the following program listing:

```
#! /usr/bin/env python

# golf_magnus.py

import numpy
import pylab, matplotlib

from numpy import pi


h = 0.01                        # Time step (s)
g = 9.8                         # Gravity (m/s^2)
dens = 1.2                      # Air density (kg/m^3)

m = 45.93e-3                    # Mass of golf ball (kg)
d = 42.67e-3                    # Ball diameter (m)
A = pi*(d/2.)**2                # Cross-sectional area of golf ball (m^2)

x0 = 0.0                        # Initial horizontal distance (m)
z0 = 0.0                        # Initial altitude (m)

V0 = 70.0                       # Launch speed (m/s)
theta0 = numpy.radians(15)      # Launch angle (radians)

u0 = V0 * numpy.cos(theta0)     # Initial horizontal speed (m/s)
w0 = V0 * numpy.sin(theta0)     # Initial vertical speed (m/s)


def get_spin_ratio(omega,V):
    R = omega*(d/2.)/V
    if R<0.05 or R>2:
        raise ValueError, 'Spin ratio outside of allowed range: %f' % (R,)
    else:
        return R


def get_Cd(omega,V):
    R = get_spin_ratio(omega,V)
    return 0.1403 - 0.3406*R*numpy.log(R) + 0.3747*R**1.5


def get_Cl(omega,V):
    R = get_spin_ratio(omega,V)
    return 0.3996 + 0.1583 * numpy.log(R) + 0.03790 * R**(-0.5)
```

```
# Perform the finite difference calculation
def get_trajectory(rpm):

    x = [x0,u0*h + x0]         # Horizontal distances (m)
    z = [z0,w0*h + z0]         # Altitudes (m)
    omega = 2*pi * rpm / 60.   # Spin rate (radians/s)

    while z[-1] > 0:

        # Get the air speeds
        u = (x[-1]-x[-2]) / h
        w = (z[-1]-z[-2]) / h
        V = numpy.sqrt(u**2 + w**2)

        # Determine the lift and drag coefficient
        Cd = get_Cd(omega,V)
        Cl = get_Cl(omega,V)

        # Determine the next point in the trajectory
        xnext = -dens*A*V/(2*m)*(Cd*u+Cl*w)*h**2 + 2*x[-1] - x[-2]
        znext = -(g+dens*A*V/(2*m)*(Cd*w-Cl*u))*h**2 + 2*z[-1] - z[-2]

        # Store the values
        x.append(xnext)
        z.append(znext)

    # Convert lists to arrays and return
    return numpy.array(x), numpy.array(z)


# Get trajectories for different rpm values
rpm1, rpm2 = 2000, 4000
x1,z1 = get_trajectory(rpm1)
x2,z2 = get_trajectory(rpm2)

# Get the trajectory without drag for comparison
x3 = numpy.arange(0,400.)
z3 = z0 + (w0/u0)*(x3-x0) - g/(2*u0**2)*(x3-x0)**2

# Select above-ground points only
x3 = x3.compress(z3>=0.)
z3 = z3.compress(z3>=0.)

# Plotting
pylab.plot(x1,z1,label='%.0f rpm'%(rpm1))
pylab.plot(x2,z2,label='%.0f rpm'%(rpm2))
pylab.plot(x3,z3,label='Vacuum')

pylab.xlabel('x')
pylab.ylabel('z')

font = matplotlib.font_manager.FontProperties(size='small')
legend = pylab.legend(loc='upper left',prop=font)
legend.draw_frame(False)

pylab.ylim(0,50)

pylab.subplots_adjust(left=0.3,right=0.7,top=0.7,bottom=0.3)
```

```
pylab.show()
```

There is something new in the `get_spin_ratio()` function: a *raise* statement. Should the function calculate a spin ratio that is outside that allowed for the USGA implementation, a `ValueError` *exception* will be raised. This completely interrupts processing in the function, and nothing is returned. For now, all we will get is an error message (like you have seen from python before for syntax errors) and our programs will exit. Later on we will learn how to handle error conditions in our programs.

Raising exceptions is the standard way of dealing with unexpected (or unwanted) conditions in a program. You can read more about what standard Exceptions are available in the python documentation at http://docs.python.org/library/exceptions.html.

The `compress()` method for arrays that is used in the vacuum trajectory calculation is also new. The trajectory itself is determined out to 400 m distance, by which point is well below ground. The `compress()` method is used to select only those points where the trajectory is above ground. Here's a simple example that demonstrates how `compress()` works:

```
>>> x = numpy.arange(6)
>>> x>=2
array([False, False,  True,  True,  True,  True], dtype=bool)
>>> x.compress(x>=2)
array([2, 3, 4, 5])
```

The `compress()` method returns values in an array according to the sequence of boolean values provided as an argument. There need to be as many boolean values as array elements for this to work.

Three trajectories determined by the program are given in Figure 4.9: two for spinning golf balls, and the vacuum case for reference. All three trajectories travel about the same distance. The vacuum case is a perfectly symmetric parabola, whereas the atmospheric trajectories are skewed, decending at a sharper angles than launched at. The golf ball that is spinning faster climbs higher. Each of these characteristics are as expected.

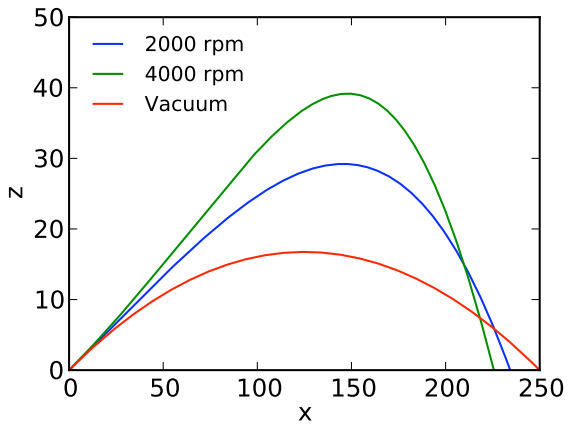

Figure 4.9: Trajectory of a spinning golf ball and the vacuum case for comparison.

4.5 Odds and Sod

There are a number of additional effects we should take into account for the most realistic simulation possible. They will be implemented together at the end of this section.

4.5.1 Horizontal Wind

Wind affects the trajectory and ultimate range of the golf ball. Suppose that there is a horizontal wind speed U. The equations of motion must be changed to account for the speed of the ball relative to wind in both the drag and Magnus forces. This is most easily accomplished by setting u, w and V as air speeds (speeds relative to air) in our calculations, rather than as the grounds speeds (speeds relative to ground) we have been using so far.

Tests with the model show that the golf ball does not travel as far into a headwind, and it lands further down range with a tailwind. Give it a try by modifying the model on your own.

4.5.2 Spin Decay

The viscous drag of air on the golf ball will cause the spin to decay. A formula that can be used to describe the change of spin rate ω with time is

$$\dot{\omega} = -C_\omega \frac{\omega V}{r}$$

where ω_0 is the initial spin rate and C_ω is the decay constant. The equation makes intuitive sense: the faster the spin or airspeed, the more the spin rate will decrease per unit time.

The analytical solution to this equation is

$$\omega = \omega_0 \exp\left(\frac{-C_\omega s}{r}\right)$$

where ω_0 is the initial spin rate and

$$s = \int V dt$$

is the path length relative to air.

How can we perform an integration numerically? Recall that an integral is just the area under a curve. The simplest approach (and there are many) is to assume a straight line between adjacent values of V (see Figure 4.10).

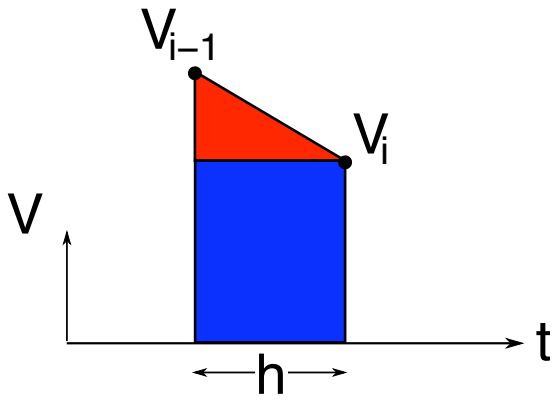

Figure 4.10: Trapezoidal integration. The area under the curve is the sum of the areas of the triangle and rectangle.

The area under the line segment between the two points is given by

$$V_i h + \frac{1}{2}(V_i - V_{i-1})h$$

where h is the time step, as usual. To integrate an entire curve, sum up the area of all of the trapezoids between adjacent points. This is called *trapezoidal integration*.

4.5.3 Carry

The *carry* of a golf ball describes the horizontal distance it travels between launch and landing. To accurately determine the carry, we need to know exactly where the ball hits ground. In our previous work we continued the calculations until the ball was below zero altitude. In the future we will allow for topography, and so the trajectory calculations will be continued until the ball is below ground (whatever altitude that happens to be). Since the calculation has proceeded too far in any case, we need to determine where the trajectory intersects with ground to obtain the best carry estimate.

We know that the trajectory will cross ground in the last two points, so let's start by considering the equation for a straight-line segment between them. The equations for a line is

$$z = mx + b$$

where m is the slope and b is the z-intercept. Using the last two points on the trajectory at indices N-1 and N-2 gives

$$z_{N-1} = mx_{N-1} + b$$

and

$$z_{N-2} = mx_{N-2} + b \quad .$$

Solving for the slope and z-intercept yields

$$m = \frac{z_{N-1} - z_{N-2}}{x_{N-1} - x_{N-2}}$$

and

$$b = z_{N-1}x_{N-2} - z_{N-2}x_{N-1}x_{N-2} - x_{N-1}$$

respectively. The first step in finding where the ball lands is to calculate `m` and `b`.

Now, suppose that the ground altitude is specified by some function `get_zground()`, which has the position `x` as a single parameter. The distance of the ball from ground at any given `x` along the straight-line segment is given by the function:

```
def dist(x):
    return (m*x+b) - get_zground(x)
```

To find where the ball strikes ground, we need to determine where `dist()` evaluates to zero. `scipy` has a handy function for that in the `optimize` submodule called `brentq()`. Here's how to use it:

```
xend = scipy.optimize.brentq(dist, xmin, xmax, xtol=0.1)
```

The first argument gives the function we wish to find the zero of, the next two arguments give the minimum and maximum possible values for x, and the final one gives what we will tolerate for error in the final answer (here 0.1 m).

`scipy.optimize.brentq()` implements Brent's Algorithm, a version of which we will explain in the next Chapter. For now it is enough for us to know that it returns the location of the zeros of a function (in this case `dist()`) given a suitable bracket (`xmin` and `xmax`) and tolerance (0.1).

A more compact way to write the above code is:

```
xend = scipy.optimize.brentq(lambda x: (m*x+b)-get_zground(x),
                             xmin, xmax, xtol=0.1)
```

Here we have used a so-called *lambda function*, which is defined by the expression after the colon. Lambda functions are useful in that they are very compact and require no name.

4.5.4 Roll

The carry is augmented by the *roll* of the ball after landing. The roll depends strongly on the angle of incidence between the trajectory and the ground: the smaller the angle the longer the roll. A simple but reasonable parameterization for the roll (in m) on a fairway after a drive is

$$x_{roll} = 17.56 - 1.035\sqrt{|a| - 35}$$

where the angle a is measured in degrees (Personal communication from Steve Quintavalla, USGA).

4.5.5 Implementation and Code Optimization

We already have a considerable body of code, and will want to use it for a variety of different calculations. To that end, we carve off most of it into the :mod:golf: module, as shown below:

```
# golf.py - a module for solving the golf problem

import numpy
import scipy.optimize

from numpy import pi


h = 0.01           # Time step (s)
g = 9.8            # Gravity (m/s^2)
dens = 1.2         # Air density (kg/m^3)

m = 45.93e-3       # Mass of golf ball (kg)
d = 42.67e-3       # Ball diameter (m)
A = pi*(d/2.)**2   # Cross-sectional area of golf ball (m^2)

Nstore = int(round(0.5/h))   # Store every Nth position in get_trajectory


def get_spin_ratio(omega, V):
    """Returns the spin ratio.
```

```
    A ValueError is raised if the spin ratio is outside the range allowed
    by the lift and drag parameterizations.
    """
    R = omega*(d/2.)/V
    if R<0.05 or R>2:
        raise ValueError, 'Spin ratio outside of allowed range: %f' % R
    else:
        return R


def get_Cd(omega, V):
    """Returns the drag coefficient Cd

    The parameterization from the USGA Second Report on the Study of
    Spin Generation (2007, Appendix C) is used.
    """
    R = get_spin_ratio(omega, V)
    return 0.1403 - 0.3406*R*numpy.log(R) + 0.3747*R**1.5


def get_Cl(omega, V):
    """Returns the lift coefficient Cl.

    The parameterization from the USGA Second Report on the Study of
    Spin Generation (2007, Appendix C) is used.
    """
    R = get_spin_ratio(omega, V)
    return 0.3996 + 0.1583 * numpy.log(R) + 0.03790 * R**(-0.5)


def get_U_default(x,z):
    return 0.


def get_zground_default(x):
    return 0.


# Perform the finite difference calculation
def get_trajectory(V0, theta0, rpm0, x0=0, z0=0,
                   get_zground=get_zground_default,
                   get_U=get_U_default,
                   N=100):
    """Returns the trajectory x,z coordinates for a golf ball.

    V0 - the launch speed (m/s)
    theta0 - the launch angle (degrees)
    rpm0 - the rotation rate of the ball at launch (revolutions per minute)
    x0 - initial horizontal distance (m)
    z0 - initial altitude (m)
    get_zground(x) - a function that returns the ground altitude (m) for
                     position x
    get_U(x,z) - a function that returns the eastward wind speed (m/s) for
                 position x,z
    N - the maximum number of points to return in the trajectory
    """
```

```
theta0 = numpy.radians(theta0)    # Launch angle (radians)

u0 = V0 * numpy.cos(theta0)   # Initial horizontal speed (m/s)
w0 = V0 * numpy.sin(theta0)   # Initial vertical speed (m/s)

# Initialize previous and current values for x, z and V
xprev, xcur = x0, u0*h + x0
zprev, zcur = z0, w0*h + z0
Vprev, Vnow = V0, None

# Create arrays to store the trajectory, and save the first point
x = numpy.zeros(N)                          # Horizontal distances (m)
z = numpy.zeros(N)                          # Altitudes (m)
filled = numpy.zeros(N,dtype=numpy.bool)  # Flags points filled in
x[0], z[0], filled[0] = xprev, zprev, True

# Determine the number of steps between save points.  Assume flight
# times are up to 12 seconds long.
step = int(numpy.ceil(round(12./h)/N))

# Initial spin rate in radians/s
omega0 = 2*pi * rpm0 / 60.

# Glob some constants together
B = -dens*A/(2*m)*h**2

# Set the initial path length to zero
s = 0.

i = 2
while zcur >= get_zground(xcur) and i//step+2<N:

    # Get the air speeds
    u = (xcur-xprev) / h - get_U(xcur,zcur)
    w = (zcur-zprev) / h
    Vcur = numpy.sqrt(u**2 + w**2)

    # Determine the path length and spin rate
    s += Vcur*h + 0.5*(Vcur-Vprev)*h  # Trapezoidal integration
    omega = omega0 * numpy.exp( -2.e-5 * s / (d/2) )

    # Determine the lift and drag coefficients
    Cd = get_Cd(omega,Vcur)
    Cl = get_Cl(omega,Vcur)

    # Determine the next point in the trajectory
    xnext = B*Vcur*(Cd*u+Cl*w) + 2*xcur - xprev
    znext = -g*h**2 + B*Vcur*(Cd*w-Cl*u) + 2*zcur - zprev

    # Save trajectory points as requested
    if znext > get_zground(xcur) or i//step == 0:
        x[i//step+1] = xnext
        z[i//step+1] = znext
        filled[i//step+1] = True
    else: # Don't overwrite the second last point
        x[i//step+2] = xnext
        z[i//step+2] = znext
        filled[i//step+2] = True
```

```
        # Update the previous and current values for x, z and V
        xprev, xcur = xcur, xnext
        zprev, zcur = zcur, znext
        Vprev, Vcur = Vcur, None

        i = i+1  # Increment counter

    # Check that the calculation completed
    if i//step+2 >= N:
        raise RuntimeError, 'Trajectory longer than 12 seconds'

    # Return compressed arrays
    return x.compress(filled), z.compress(filled)


def get_carry_distance(x,z,get_zground=get_zground_default):
    """Returns the horizonal distance (m) traveled in-flight.

    x - the horizontal distances (m) of the trajectory
    z - the corresponding altitudes (m) of the trajectory
    get_zground(x) - a function that returns the ground altitude (m) for
                     position x
    """

    if x[-1]==x[-2]:     # Special case: coming straight down
        return x[-1]

    else:
        # Find z=mx+b between last two points in trajectory
        m = (z[-1]-z[-2])/(x[-1]-x[-2])
        b  = (z[-1]*x[-2]-z[-2]*x[-1]) / (x[-2]-x[-1])

        # Determine where the trajectory hits ground
        xend = scipy.optimize.brentq(lambda x: (m*x+b)-get_zground(x),
                                     x[-2], x[-1], xtol=0.1)

        return numpy.fabs(x[0]-xend)


def get_roll_distance(x,z):
    """Returns the horizontal distance (m) rolled after landing.

    x - the horizontal distances (m) of the trajectory
    z - the corresponding altitudes (m) of the trajectory

    The parameterization was provided in a personnal communication from
    Steve Quintavalla of the USGA.

    It is assumed that the fairway is flat.
    """

    # Determine the angle of incidence
    dx = x[-1]-x[-2]
    dz = z[-2]-z[-1]
    angle = numpy.arcsin(dz/numpy.sqrt(dx**2+dz**2))

    return (17.56 - 1.035 * (numpy.degrees(angle) - 35))
```

```
def get_total_distance(x,z):
    """Returns the total distance traveled: trajectory carry + roll.

    x - the horizontal distances (m) of the trajectory
    z - the corresponding altitudes (m) of the trajectory
    """
    return get_carry_distance(x,z) + get_roll_distance(x,z)
```

There are a few changes from what we saw before, including some new functions based on the descriptions in the previous subsections. All of the parameters we expect to keep constant are defined up front. We will want to vary many of the others, and so they have been added as arguments to the `get_trajectory()` function. We will need to set `V0`, `theta0` and `rpm0` every time, so they are given as the first three arguments. The other arguments have all been assigned *default values*. If we choose not to set them in a function call, the defaults will be used. An example call of this type is:

```
golf.get_trajectory(70,15,2000)
```

The remaining arguments may be provided in order, or by using keywords. For example, to use a starting altitude of 10 m but leave everything else as default, write:

```
golf.get_trajectory(70,15,2000,z0=10)
```

Notice that the `get_trajectory()` accepts functions for the `get_zground` and `get_U` parameters, which are used to define the ground altitude and wind speed, respectively. We have assigned simple defaults to these arguments for now: the `get_U_default()` function returns a wind speed of 0 m/s, and the `get_zground_default()` returns an altitude of 0 m. We can define more complicated wind profiles and topographies in the future.

A few things are different in the `get_trajectory()` internals. We don't need to keep all of the points on a trajectory. Instead of storing *everything* in lists, we keep track of the previous, current, and next values, and only store the occasional trajectory point in an array. The array is of fixed length, and we perform some checks to make sure we don't exceed that (and raise an Exception if we do). Also, to make the calculation more efficient, we glob some of the constants together so that we aren't doing the exact same multiplications every iteration in the loop.

The loop in the `get_trajectory()` function is well suited for implementation in C, which will make it much faster. Let's create a new module, `cgolf`, and use that in the future:

```
import numpy
import scipy.weave

from golf import *

def testfunc(s=None):
    print s

# Perform the finite difference calculation
def get_trajectory(V0, theta0, rpm0, x0=0, z0=0,
                   get_zground=get_zground_default,
                   get_U=get_U_default,
                   N=100):
    """Returns the trajectory x,z coordinates for a golf ball.

    V0 - the launch speed (m/s)
    theta0 - the launch angle (degrees)
    rpm0 - the rotation rate of the ball at launch (revolutions per minute)
    x0 - initial horizontal distance (m)
```

```
    z0 - initial altitude (m)
    get_zground(x) - a function that returns the ground altitude (m) for
                     position x
    get_U(x,z) - a function that returns the eastward wind speed (m/s) for
                 position x,z
    N - the maximum number of points to return in the trajectory
    """

    theta0 = numpy.radians(theta0)     # Launch angle (radians)

    u0 = V0 * numpy.cos(theta0)    # Initial horizontal speed (m/s)
    w0 = V0 * numpy.sin(theta0)    # Initial vertical speed (m/s)

    # Initialize previous and current values for x, z and V
    xprev, xcur = float(x0), float(u0*h + x0)
    zprev, zcur = float(z0), float(w0*h + z0)
    Vprev = float(V0)

    # Create arrays to store the trajectory, and save the first point
    x = numpy.zeros(N)                          # Horizontal distances (m)
    z = numpy.zeros(N)                          # Altitudes (m)
    filled = numpy.zeros(N,dtype=numpy.bool)   # Flags points filled in
    x[0], z[0], filled[0] = xprev, zprev, True

    # Determine the number of steps between save points.  Assume flight
    # times are up to 12 seconds long.
    step = int(numpy.ceil(round(12./h)/N))

    # Initial spin rate in radians/s
    omega0 = 2*pi * rpm0 / 60.

    # Glob some constants together
    B = -dens*A/(2*m)*h**2

    code = r"""
#include <math.h>

PyObject *args, *result;  /* Used for python function calls */

int i = 2;           /* Time index */
double zground;      /* Current ground altitude */
double U;            /* Current wind speed */
double u, w, Vcur;   /* Air speeds */
double s = 0.;       /* Air path length */
double omega;        /* Current spin rate */
double R;            /* Sping ratio */
double Cd, Cl;       /* Drag and lift coefficients */
double xnext,znext;  /* The next trajectory position */
double tmp;

/* Get the current ground altitude (from the python function) */
args = Py_BuildValue("(f)",xcur);
result = PyEval_CallObject(get_zground,args);
if(!PyFloat_Check(result)) {
  PyErr_SetString(PyExc_TypeError,"get_zground() should return a float");
  return NULL;
}
zground = PyFloat_AsDouble(result);
```

```
Py_DECREF(args);

while(zcur >= zground && i/step+2<N) {

  /* Get the current wind speed (from the python function) */
  args = Py_BuildValue("(f,f)",xcur,zcur);
  result = PyEval_CallObject(get_U,args);
  if(!PyFloat_Check(result)) {
    PyErr_SetString(PyExc_TypeError,"get_U() should return a float");
    return NULL;
  }
  U = PyFloat_AsDouble(result);
  Py_DECREF(args);

  /* Get the air speeds */
  u = (xcur-xprev) / h - U;
  w = (zcur-zprev) / h;
  Vcur = sqrt(u*u + w*w);

  /* Determine the path length and spin rate */
  s += Vcur*h + 0.5*(Vcur-Vprev)*h;  /* Trapezoidal integration */
  omega = ((double)omega0) * exp( -2.e-5 * s / (d/2) );

  /* Determine the lift and drag coefficients */
  R = omega*(d/2)/Vcur;
  if(R<0.05 or R>2) {
    PyErr_SetString(PyExc_ValueError, "Spin ratio outside of allowed range");
    return NULL;
  }
  Cd = 0.1403 - 0.3406*R*log(R) + 0.3747*pow(R,1.5);
  Cl = 0.3996 + 0.1583 * log(R) + 0.03790 * pow(R,-0.5);

  /* Determine the next point in the trajectory */
  xnext = B*Vcur*(Cd*u+Cl*w) + 2*xcur - xprev;
  znext = -g*h*h + B*Vcur*(Cd*w-Cl*u) + 2*zcur - zprev;

  /* Update zground (using the python function) */
  args = Py_BuildValue("(f)",xcur);
  result = PyEval_CallObject(get_zground,args);
  if(!PyFloat_Check(result)) {
    PyErr_SetString(PyExc_TypeError,"get_zground() should return a float");
    return NULL;
  }
  zground = PyFloat_AsDouble(result);

  /* Save trajectory points as requested; Don't overwrite the second
   * last point
   */
  if(znext > zground or i%step==0) {
    x(i/step+1) = xnext;
    z(i/step+1) = znext;
    filled(i/step+1) = 1;
  }
  else {
    x(i/step+2) = xnext;
    z(i/step+2) = znext;
    filled(i/step+2) = 1;
  }
```

```
  /* Update the previous and current values for x, z and V */
  xprev = xcur;
  xcur = xnext;
  zprev = zcur;
  zcur = znext;
  Vprev = Vcur;

  i = i+1;  /* Increment counter */
}

/* Check that the calculation completed */
if(i/step+2 >= N) {
  PyErr_SetString(PyExc_RuntimeError, "Trajectory longer than 12 seconds");
  return NULL;
}
"""

    scipy.weave.inline(code,['xprev', 'xcur', 'zprev', 'zcur', 'Vprev',
                             'x','z','filled',
                             'step', 'omega0', 'h', 'B', 'd', 'g',
                             'get_zground','get_U','N'
                             ],
                       type_converters=scipy.weave.converters.blitz)

    # Return compressed arrays
    return x.compress(filled), z.compress(filled)
```