Computational Physics With Python

Dr. Eric Ayars

California State University, Chico

Chapter 4

Ordinary Differential Equations

If you give a large wooden rabbit some initial vertical velocity v over a castle wall, you will note that the vertical component of velocity gradually decreases. Eventually the vertical velocity component becomes negative, and the wooden rabbit comes back to the ground with a speed roughly equal to the speed it had when it left.

This phenomenon of "constant-acceleration motion" was probably discussed in your introductory physics course, although less Python-oriented courses tend to use more standard projectiles. Ignoring for the moment the fact that the exact solution to constant-acceleration motion is well-known,[1] let's use numeric techniques to approximate the solution.

To find the velocity v of the projectile, we use the definition of acceleration:

$$a \equiv \frac{dv}{dt}$$

Rearranging this a bit gives us

$$dv = a\,dt$$

or since we're going to need to work with finite-sized steps,

$$\Delta v = a\Delta t$$

Knowing the initial velocity v_i and the change Δv in the velocity we can estimate the new velocity:

$$v = v_i + a\Delta t \Longrightarrow v_{i+1} = v_i + \frac{d}{dt}\left[v\right]\Delta t \tag{4.1}$$

[1] $v(t) = v_i + at$, $x(t) = x_i + v_i t + \frac{1}{2}at^2$

We can use an identical process to estimate the position of the projectile from its current state $\{x_i, v_i\}$ and the definition $v = \frac{dx}{dt}$:

$$x_{i+1} = x_i + \frac{d}{dt}[x]\Delta t \tag{4.2}$$

This method does not give an exact result, but for small Δt it's reasonably close. It may seem ridiculous to use it, given that we know the answer in exact form already, but for many problems we do *not* know the exact form of the answer. If we consider air resistance, for example, then the acceleration of a projectile depends on the temperature, altitude, and projectile velocity. In such a case, we may have no option other than this sort of approximation.

4.0 Euler's Method

The method we've described so far is called *Euler's method*, and it may be helpful to consider a picture of how it works. (See figure 4.0.)

Starting from point (t_1, x_1), take the slope of $x(t)$ at that point and follow the slope through the time-step Δt to find the approximate value of x_2. Repeat this process from point x_2 to find x_3, and so on.

The error shown in figure 4.0 is pretty large, but you can immediately see that the results would be better with smaller Δt. To determine the sensitivity of Euler's method to the size of Δt, start with a Taylor expansion of $x(t)$,

$$x(t + \Delta t) = x(t) + \dot{x}\Delta t + \ddot{x}\frac{\Delta t^2}{2} + \cdots \tag{4.3}$$

where $\dot{x}$ indicates a first derivative with respect to time, $\ddot{x}$ is the second derivative, and so on. The first two terms on the right side of equation 4.3 are Euler's method. The error in each step for Euler's method is on the order of Δt^2, since that's the first term omitted in the Taylor expansion. However, the number of steps is $N = \tau/\Delta t$ so the total error by the end of the process is on the order of Δt. Decreasing the size of Δt improves your result linearly.

Notice that Euler's method only works on first-order differential equations. This is not a limitation, though, because higher-order differential equations can be expanded into systems of first-order differential equations.

Example 4.0.1

The equation for damped harmonic motion is given by

$$\ddot{x} = -\omega^2 x - \beta\dot{x} \tag{4.4}$$

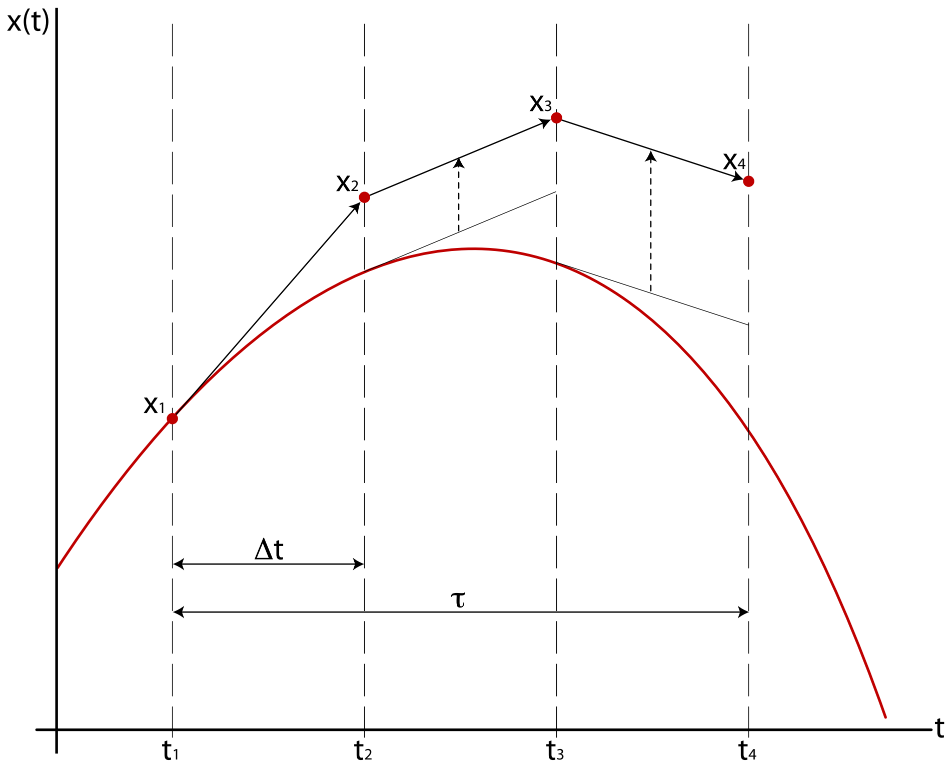

Figure 4.0: Euler's method: The slope at each point x_i is used to find the new point x_{i+1}. The smooth curve is the actual function, and the small circles are the approximation to the function determined by Euler's method.

We can express this second-order differential equation as two first-order differential equations by introducing a new variable v:

$$v \equiv \dot{x} \tag{4.5}$$

With this definition, equation 4.4 becomes

$$\dot{v} = -\omega^2 x - \beta v \tag{4.6}$$

which is a first-order differential equation.

First-order equations 4.5 and 4.6 together are equivalent to second-order equation 4.4.

4.1 Standard Method for Solving ODE's

We're going to introduce a number of methods of solving ODE's in this chapter, all of them better than Euler's method in one way or another. Before we do, though, let's take some time to develop a "standard model" for solving ODE's. That way, we can set any problem up once and then use the method of our choice to solve it, with a minimum amount of reprogramming on our part.

To start, consider the differential equation for a large wooden rabbit in free-fall:

$$\ddot{x} = -g \tag{4.7}$$

Equation 4.7 can be broken into two first-order equations:

$$\begin{aligned} \dot{x} &= v \\ \dot{v} &= -g \end{aligned} \tag{4.8}$$

The individual Euler-method solutions to those first-order equations are

$$\begin{aligned} x_{i+1} &= x_i + \dot{x}\Delta t \\ v_{i+1} &= v_i + \dot{v}\Delta t \end{aligned} \tag{4.9}$$

There is a symmetry in equations 4.9 that just makes you want to write them as a single vector equation:

$$y_{i+1} = y_i + \dot{y}\Delta t \tag{4.10}$$

where

$$y = \begin{bmatrix} x \\ v \end{bmatrix} \tag{4.11}$$

Equations 4.8 could be written with the same vector notation:

$$\dot{y} = \begin{bmatrix} v \\ -g \end{bmatrix} \tag{4.12}$$

Now here's the critical thing: *Equations 4.11 and 4.12 define the problem.* Equation 4.11 defines how things are arranged in the vector y, which holds everything we know about the system at some instant. Equation 4.12 defines the differential equation we're solving.

Now if only Python could do vector math correctly... Lists and tuples just don't work at all here, since multiplying a list by a number n just gives us a list containing n copies of the original list. But the numpy package provides just this vector-math functionality. (See Chapter 3.)

This vector notation allows us to break the process into two parts: defining the problem and solving the problem. We define the problem with a function that returns the derivatives of each element in y, as in equation 4.12.

Example 4.1.1
Define a function that will return the derivatives necessary to calculate the motion of a large wooden rabbit in free-fall.

```
def FreeFall(state, time):
    """
    This function defines the ODE d^2x/dt^2 = -g.
    It takes the vector y and returns another vector
    containing the derivatives of each element in
    the current state of the system.

    The first element in state is position x, and the
    derivative of x is velocity, v. So the first
    element in the return vector is v, which is the
    second element in state.

    The second element in state is v, and the
    derivative of v is -g for free-fall.

    The result is returned as a numpy array so the
    rest of the program can do vector math.
    """
    g0 = state[1]
    g1 = -9.8
    return numpy.array([g0, g1])
```

Notice that we've passed "time" to the function in example 4.1.1, but we haven't used it. Time does not enter into the problem, since gravitational force is a constant. We've included time since we're developing a *general* method for ODE's and our method should include capability of dealing with time-dependent equations.

Now that the problem is defined, we need a general method of calculating the next "state" of the system.

Example 4.1.2
Write a general Euler's-method routine that will calculate the next state of the system from the current state, the derivatives, and the desired time step.

```
def euler(y, t, dt, derivs):
    """
    A routine that impliments Euler's method of finding
    the new 'state' of y, given the current state, time,
    and desired time step. 'derivs' must be a function
    that returns the derivatives of y and thus defines
    the differential equation.
    """
    y_next = y + derivs(y,t) * dt
    return y_next
```

Again, this routine thinks it needs to know the time, just to ensure that the general method works for time-dependent ODE's. If time does not matter, just give it *something* to keep it happy. The Euler routine will pass the time along to the derivs function, which will ignore it if it's not needed.

Note that the code in example 4.1.1 is specific to the problem at hand, but the code in example 4.1.2 is generally applicable any time we want to use Euler's method. It would be a good idea to save the definition in 4.1.2 somewhere handy so that you can just import it when needed.[2]

In the next example, let's take everything we've done so far and put it together.

Example 4.1.3
Write a program that plots the motion of a mass oscillating at the end of a spring. The force on the mass should be given by $F = -mg + kx$.

```
#!/usr/bin/env python
"""
Program to plot the motion of a mass hanging on the
end of a spring. The force on the mass is given by
F = -mg - kx.
"""
```

[2]I personally keep a file called "tools.py" that contains all of the generally-handy functions developed in this course, such as the secant method for rootfinding and Simpson's method of integrating.

```
from pylab import *
from tools import euler # Euler's method for ODE's, written earlier

N = 1000                  # number of steps to take
xo = 0.0                  # initial position, spring
                          # unstretched.
vo = 0.0                  # initial velocity

tau = 3.0                 # total time for the
                          # simulation, in seconds.
dt = tau/float(N-1)       # time step

k = 3.5                   # spring constant, in N/m
m = 0.2                   # mass, in kg
gravity = 9.8             # g, in m/s^2

# Since we're plotting vs t, we need time for that plot.
time = linspace(0, tau, N)

"""
Create a Nx2 array for storing the results of our
calculations. Each 2-element row will be used for
the state of the system at one instant, and each
instant is separated by time dt. The first element
in each row will be position, the second velocity.
"""
y = zeros([N,2])

y[0,0] = xo               # set initial state
y[0,1] = vo

def SHO(state, time):
    """
    This defines the differential equation we're
    solving: dx^2/dt = -k/m x - g.

    We break this second-order DE into two first-
    order DE's by introducing v:
    dx/dt = v
    dv/dt = k/m x - g
    """
    g0 = state[1]
    g1 = -k/m * state[0] - gravity
    return array([g0, g1])
```

```
# Now we do the calculations.
# Loop only to N-1 so that we don't run into a
# problem addressing y[N+1] on the last point.
for j in range(N-1):
    # We give the euler routine the current state
    # y[j], the time (which we don't care about
    # at all in this ODE) the time step dt, and
    # the derivatives function SHO().
    y[j+1] = euler(y[j], time[j], dt, SHO)

# That's it for calculations! Now graph the results.
# start by pulling out what we need from y.
xdata = [y[j,0] for j in range(N)]
vdata = [y[j,1] for j in range(N)]

plot(time, xdata)
plot(time, vdata)
xlabel("time")
ylabel("position, velocity")
show()
```

The output of the program in example 4.1.3 is shown in figure 4.1. Note that the position is always negative, while the velocity is either positive or negative.

4.2 Problems with Euler's Method

Euler's method is easy to understand, but it has one very large problem. Since the method approximates the solution as a linear equation, the Euler solution always underestimates the curvature of the solution. The result is that for any sort of oscillatory motion, the energy of Euler's solution increases with time.

If you look closely at figure 4.1, you can see that the maximum velocity is increasing even after just two cycles. If we change the total simulation time tau to 20 seconds in the example 4.1.3, the error is more visible: it grows exponentially as shown in figure 4.2. Increasing the number of steps improves things, (Figure 4.3) but the error is still there and still increases exponentially with each step. Even increasing the number of steps to absurd levels does not eliminate the problem: at 200,000 points there is still a slight but visible increase in peak velocity in the output of the program!

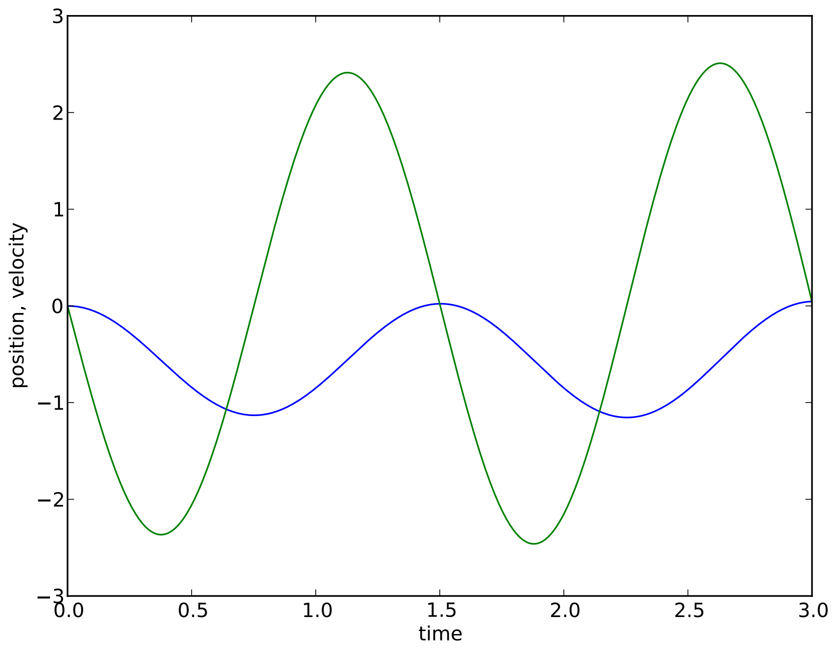

Figure 4.1: Output of program given in example 4.1.3.

4.3 Euler-Cromer Method

There is a very simple fix to Euler's method that causes it to work much better for oscillatory systems.[3] The "stock" Euler method uses the position x_i and velocity $\dot{x}_i$ at step i to find the position x_{i+1}. As it turns out, the errors exactly cancel out — for simple harmonic motion — if instead of x_i and $\dot{x}_i$ we use x_i and $\dot{x}_{i+1}$. In other words, use the current position and the *next* velocity to find the next position.

Although this is a clever fix for the current problem, we have to be a bit suspicious of it. The Euler-Cromer method gives good results for simple harmonic motion, but it's not immediately obvious that it would give the same improvment for other ODE's for which we don't already know the solution. We're better off developing methods of solving ODE's that give better results in general, rather than tweaking the broken Euler method so that it gives correct results for some particular problem.

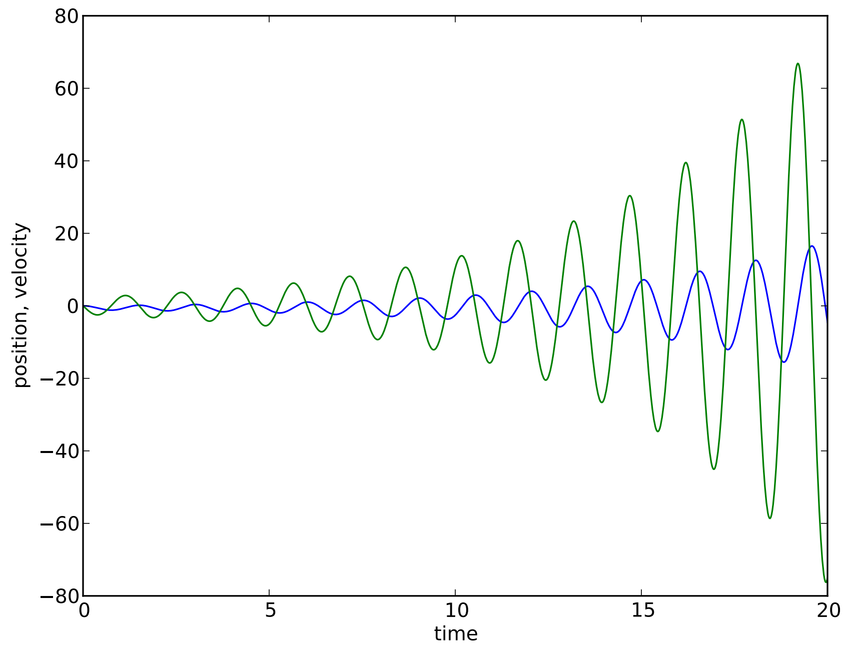

Figure 4.2: Output of program in example 4.1.3, with total time 20.0 seconds and N = 1000. Note the vertical scale!

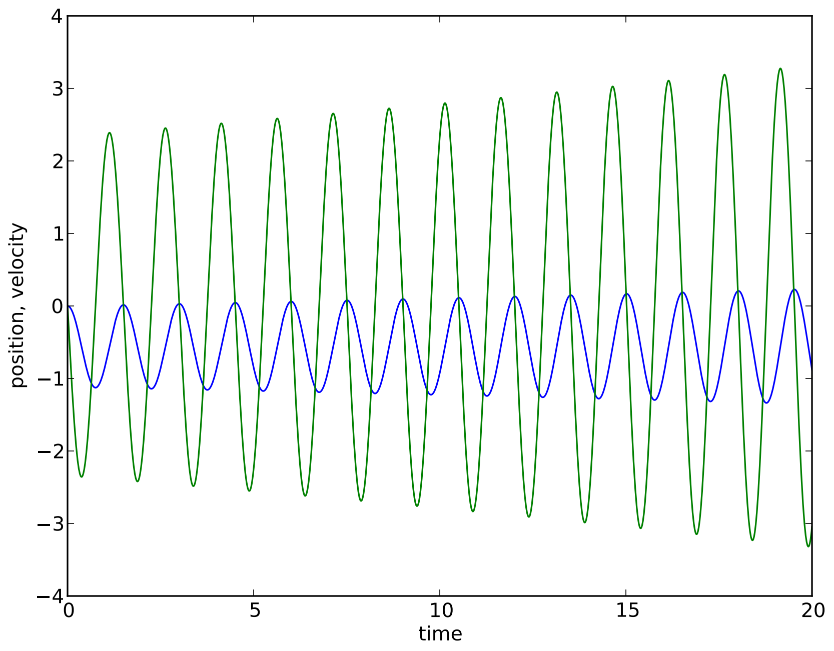

Figure 4.3: Output of program in example 4.1.3, with total time 20.0 seconds and N changed to 10,000. Despite the very large number of points in this approximation, the energy is still visibly increasing with time.

4.4 Runge-Kutta Methods

The most popular and stable general technique of solving ODE's is a set of methods known as "Runge-Kutta" methods.

I'll start the explanation of these methods with a bit of notation. We define y as the function we're looking for, and g as the derivative of y. Now g is in general a function of both y and t, so the second derivative of y is, using the chain rule,

$$\begin{aligned}\ddot{y} &= \frac{d}{dt}[\dot{y}] \\ &= \frac{d}{dt}[g(y,t)] \\ &= \frac{\partial g}{\partial t} + \frac{\partial g}{\partial y}\frac{dy}{dt} \\ &= g_t + g_y g\end{aligned}$$

where $g_\xi \equiv \frac{\partial g}{\partial \xi}$. Similarly,

$$\dddot{y} = g_{tt} + 2g g_{ty} + g^2 g_{yy} + g g_y^2 + g_t g_y$$

The Taylor expansion of $y(t + \Delta t)$ is then

$$\begin{aligned}y(t+\Delta t) &= y(t) \\ &+ g\Delta t \\ &+ \frac{\Delta t^2}{2!}(g_t + g_y g) \\ &+ \frac{\Delta t^3}{3!}(g_{tt} + 2g g_{ty} + g^2 g_{yy} + g g_y^2 + g_t g_y) \\ &+ \mathcal{O}(\Delta t^4)\end{aligned} \tag{4.13}$$

We could also expand $y(t)$ as a linear combination of some set of polynomials:

$$y(t+\Delta t) = y(t) + \alpha_1 k_1 + \alpha_2 k_2 + \cdots + \alpha_n k_n \tag{4.14}$$

where

$$\begin{aligned}
k_1 &= \Delta t g(y,t) \\
k_2 &= \Delta t g(y+\nu_{21}k_1, t+\nu_{21}\Delta t) \\
k_3 &= \Delta t g(y+\nu_{31}k_1+\nu_{32}k_2, t+\nu_{31}\Delta t+\nu_{32}\Delta t) \\
&\vdots \\
k_n &= \Delta t g\left(y+\sum_{\ell=1}^{n-1}\nu_{n\ell}k_\ell, t+\Delta t\sum_{\ell=1}^{n-1}\nu_{n\ell}\right) \qquad (4.15)
\end{aligned}$$

The α and ν are "parameters to be determined".

The next step is to expand 4.14 with a Taylor expansion, and compare the result with 4.13.

$$\begin{aligned}
y(t+\Delta t) &= y+\alpha_1 k_1+\alpha_2 k_2 \\
&= y+\alpha_1[\Delta t g(y,t)]+\alpha_2[\Delta t g(y+\nu_{21}k_1), t+\nu_{21}\Delta t] \qquad (4.16)
\end{aligned}$$

The first and second terms in 4.16 are exact already, so we expand only the third term:

$$\begin{aligned}
k_2 &= \Delta t[g+\nu_{21}k_1 g_y+\nu_{21}\Delta t g_t+\mathcal{O}(\Delta t^2)] \\
&= \Delta t g+\nu_{21}\Delta t^2 g g_y+\nu_{21}\Delta t^2 g_t+\mathcal{O}(\Delta t^3) \qquad (4.17)
\end{aligned}$$

Substituting 4.17 into 4.16 and rearranging terms gives us

$$y(t+\Delta t) = y+[(\alpha_1+\alpha_2)g]\Delta t+[\alpha_2\nu_{21}(g g_y+g_t)]\Delta t^2 \qquad (4.18)$$

We can compare 4.18 with 4.13 to determine our "parameters to be determined".

$$\begin{aligned}
\alpha_1+\alpha_2 &= 1 \\
\alpha_2\nu_{21} &= \frac{1}{2} \qquad (4.19)
\end{aligned}$$

There are three unknowns in 4.19, and only two equations, so the system is underspecified. We can pick one value, then. If we choose $\nu_{21}=1$, then α_1 and α_2 are each $\frac{1}{2}$. These values, applied to equation 4.14, give us *a* second-order Runge-Kutta method:

$$y(t+\Delta t) = y+\frac{1}{2}k_1+\frac{1}{2}k_2+\mathcal{O}(\Delta t^3) \qquad (4.20)$$

where

$$k_1 = \Delta t g(y, t)$$
$$k_2 = \Delta t g(y + k_1, t + \Delta t)$$

Conceptually, this second-order Runge-Kutta method is equivalent to taking the *average* of the slope at t and at $t + \Delta t$ and using that average slope in Euler's method to find the new value $y(t + \Delta t)$.

The error in second-order Runge-Kutta method is on the order of Δt^3 per step, so after $N = \frac{\tau}{\Delta t}$ steps the total error is on the order of Δt^2. This total error is a significant improvement over Euler's method, even considering the roughly 2× increase in computational effort. If $\frac{\Delta t}{\tau} \approx 0.01$, then the error (compared to Euler's method) goes down by about 100, for a cost increase of about 2. Alternately, one can use this method to get the *same* accuracy as with Euler's method for a fifth of the effort.

An important consideration is that the solution to 4.19 given by 4.20 is *one* of the possible second-order Runge-Kutta methods. We could just as easily set $\alpha_1 = 0$ in 4.19, in which case $\alpha_2 = 1$ and $\nu_{21} = \frac{1}{2}$. In that case, the resulting second-order method becomes

$$y(t + \Delta t) = y(t) + \Delta t g(y + \frac{1}{2}k_1, t + \frac{1}{2}\Delta t) \tag{4.21}$$

where $k_1 = \Delta t g(y, t)$. This second-order Runge-Kutta method is equivalent to taking the slope at the midpoint between t and $t + \Delta t$, and using that slope in Euler's method.

Example 4.4.1
Here is a second-order Runge-Kutta method routine:

```
def rk2(y, time, dt, derivs):
    """
    This function moves the value of 'y' forward by a single
    step of size 'dt', using a second-order Runge-Kutta
    algorithm. This particular algorithm is equivalent to
    finding the average of the slope at time t and at time
    (t+dt), and using that average slope to find the new
    value of y.
    """
    k0 = dt * derivs(y, time)
    k1 = dt * derivs(y + k0, time + dt)
    y_next = y + 0.5 * (k0 + k1)

    return y_next
```

It's a worthwhile exercise to modify the code in example 4.1.3 to use this routine instead of the Euler method and see the difference. The effort we put into splitting the definition of the differential equation from the solution method, in section 4.1, pays off now: if you add the function rk2() to your 'tools.py' file, the only code change necessary to make this switch is to import and call 'rk2' instead of 'euler'.

This second-order Runge-Kutta method comes from taking a Taylor expansion of the function we're trying to solve, and then keeping the first few terms in that expansion. As you might guess from previous sections in this book, it's possible to take more terms and get even better accuracy. Using the same procedure outlined above — but taking terms to fourth order — gives us another set of equations involving α and ν, which are underspecified just as in the case of second-order Runge-Kutta. The most commonly-used solution generates this set of equations:

$$\begin{aligned} y\left(t+\Delta t\right) &= y\left(t\right)+\frac{1}{6}\left(k_1+2k_2+2k_3+k_4\right)+\mathcal{O}\left(\Delta t^4\right) \\ k_1 &= g\left(y,t\right)\Delta t \\ k_2 &= g\left(y+\frac{1}{2}k_1, t+\frac{1}{2}\Delta t\right)\Delta t \\ k_3 &= g\left(y+\frac{1}{2}k_2, t+\frac{1}{2}\Delta t\right)\Delta t \\ k_4 &= g\left(y+k_3, t+\Delta t\right)\Delta t \end{aligned} \tag{4.22}$$

Converting this algorithm to a useful function is left as an exercise to the student. (It's not particularly difficult.)

There are more specialized methods of solving differential equations which can give better results for certain problems, but as a general-purpose method fourth-order Runge-Kutta offers a good combination of speed, stability, and precision. There is one significant area of improvement for this method, *adaptive stepsize*, which you'll have to learn from a different source.[13]

Here is an example of solving a differential equation in which the force changes direction periodically.

Example 4.4.2

A spring and mass system is shown in figure 4.4. The coefficient of friction μ is not negligible. Generate a position vs. time plot for the motion of the mass, given an initial displacement $x = 0.2$

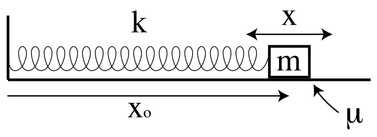

Figure 4.4: Horizontal Spring/mass system

m, spring constant $k = 42$ N/m, mass $m = 0.25$ kg, coefficient of friction $\mu = 0.15$, and initial velocity $v_i = 0$.

Most of the solution is identical to the program in example 4.1.3. There are minor changes required, such as defining μ, adjusting parameters, and changing the solving function from euler() to rk4(), but the significant change is replacing the derivative function SHO() with a new function specific to this problem.

```
#!/usr/bin/env python
"""
Program to plot the motion of a mass and spring on
a horizontal surface with friction.
F = - kx +/- mu m g
"""

from pylab import *
from tools import rk4

N = 500                   # number of steps to take
xo = 0.2                  # initial position
vo = 0.0                  # initial velocity

tau = 3.0                 # total time for the
                          # simulation, in seconds.
dt = tau/float(N-1)       # time step

k = 42.0                  # spring constant, in N/m
m = 0.25                  # mass, in kg
gravity = 9.8             # g, in m/s^2
mu = 0.15                 # friction coefficient

"""
Create a Nx2 array for storing the results of our
calculations. Each 2-element row will be used for
```

```
the state of the system at one instant, and each
instant is separated by time dt. The first element
in each row will be position, the second velocity.
"""
y = zeros([N,2])

y[0,0] = xo              # set initial state
y[0,1] = vo

def SpringMass(state, time):
    """
    This defines the differential equation we're
    solving: dx^2/dt = -k/m x +/- mu g.

    We break this second-order DE into two first-
    order DE's by introducing v:
    dx/dt = v
    dv/dt = k/m x +/- mu g

    Note that the direction of the frictional
    force changes, depending on the sign of v.
    We handle this with an if statement.
    """
    g0 = state[1]

    if g0 > 0:
        g1 = -k/m * state[0] - gravity * mu
    else:
        g1 = -k/m * state[0] + gravity * mu

    return array([g0, g1])

# Now we do the calculations.
# Loop only to N-1 so that we don't run into a
# problem addressing y[N+1] on the last point.
for j in range(N-1):
    # We give the euler routine the current state
    # y[j], the time (which we don't care about
    # at all in this ODE) the time step dt, and
    # the derivatives function SHO().
    y[j+1] = rk4(y[j], 0, dt, SpringMass)

# That's it for calculations! Now graph the results.
time = linspace(0, tau, N)
plot(time, y[:,0], 'b-', label="position")
```

```
xlabel("time")
ylabel("position")
show()
```

Notice that the derivs() function can contain anything we need to make things work. In this case, the function uses an **if** statement to change the direction of the frictional force as needed. The graphical output of the complete program is shown in figure 4.5.

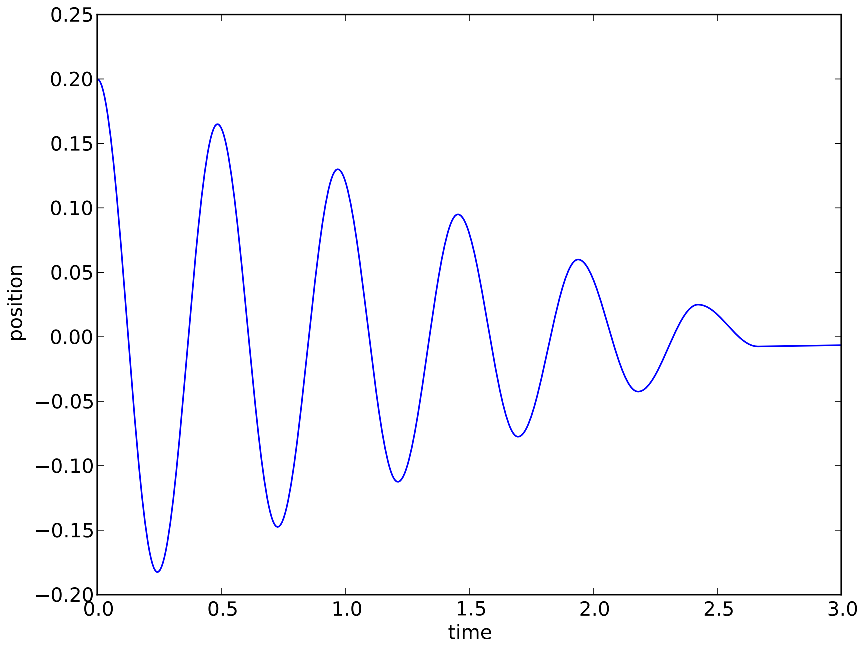

Figure 4.5: Motion of mass shown in figure 4.4

There is one more level of improvement we can make to our ODE-solving method. So far we've dealt with individual steps, but in most computational work it's more common to find solutions for the differential equation at an array of times. For example, in the previous problem we could have defined an array of times for plotting (using linspace()) and then called a function that used our ODE solver method of choice to find solutions at those times. Development of such a function is left as an exercise.

4.5 Scipy

As you might expect, Scipy has a routine that solves differential equations. This function is avalable as scipy.integrate.odeint(), and it uses variable step-sizes and error-checking methods to return very precise results, efficiently. Call this routine with a derivs function, an initial state value (which may be an array, as usual) and an array of times (rather than a time step.) The odeint() function will return an array of state values corresponding to those times.

Let's use odeint() to look at a more complex example:

Example 4.5.1
A mass m is attached to a spring with spring constant k, which is attached to a support point as shown in figure 4.6. The length of the resulting pendulum at any given time is the spring rest-length x_o plus the stretch (or compression) x, and the angle of the pendulum with respect to the vertical is θ. This is an example

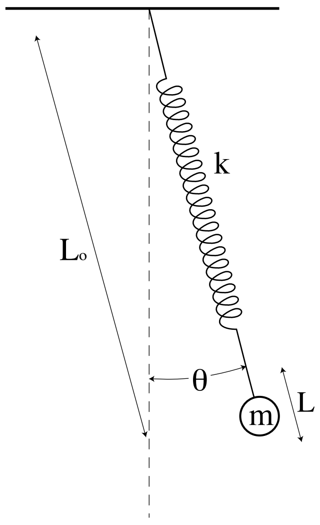

Figure 4.6: Springy Pendulum for example 4.5.1

of a coupled oscillator system: the "pendulum" oscillations in θ interact with the "spring" oscillations in x, producing a complex mix of both. The differential equations for this system are given by[3]

$$\ddot{L} = (L_o + L)\dot{\theta}^2 - \frac{k}{m}L + g\cos\theta$$
$$\ddot{\theta} = -\frac{1}{L_o + L}\left[g\sin\theta + 2\dot{L}\dot{\theta}\right]$$

Write a program that plots the motion of the mass for some initial $\theta \neq 0$.

```
#!/usr/bin/env python
"""
Program to plot the motion of a "springy pendulum".
"""

from pylab import *
from scipy.integrate import odeint

N = 1000                    # number of steps to take

"""
We actually have FOUR parameters to track, here:
    L, L_dot, theta, and theta_dot.
So instead of the usual Nx2 array, make it Nx4.
Each 4-element row will be used for the state of
the system at one instant, and each instant is
separated by time dt. I'll use the order given above.
"""
y = zeros([4])

L_o = 1.0                   # unstretched spring length
L = 1.0                     # Initial stretch of spring
v_o = 0.0                   # initial velocity
theta_o = 0.3               # radians
omega_o = 0.0               # initial angular velocity

y[0] = L                    # set initial state
y[1] = v_o
y[2] = theta_o
y[3] = omega_o

time = linspace(0, 25, N)
```

[3]The easiest way of deriving these is via Lagrangian Dynamics, see [7].

```
k = 3.5                        # spring constant, in N/m
m = 0.2                        # mass, in kg
gravity = 9.8                  # g, in m/s^2

def spring_pendulum(y, time):
    """
    This defines the set of differential equations
    we are solving. Note that there are more than
    just the usual two derivatives!
    """
    g0 = y[1]
    g1 = (L_o+y[0])*y[3]*y[3] - k/m*y[0] + gravity*cos(y[2])
    g2 = y[3]
    g3 = -(gravity*sin(y[2]) + 2.0*y[1]*y[3])/(L_o + y[0])

    return array([g0, g1, g2, g3])

# Now we do the calculations.
answer = odeint(spring_pendulum, y, time)

# Now graph the results.
# rather than graph in terms of t, I'm going
# to graph the track the mass takes in 2D.
# This will require that I change L,theta data
# to x,y data.
xdata = ( L_o + answer[:,0])*sin(answer[:,2])
ydata = -(L_o + answer[:,0])*cos(answer[:,2])

plot(xdata,ydata, 'r-')
xlabel("Horizontal position")
ylabel("Vertical position")
show()
```

Note that in previous examples the size of our state vector y has always been 2, but there's nothing magic about that. We can make it contain as many (or as few) parameters as necessary to solve the problem.

The path of the pendulum for one set of initial conditions is shown in figure 4.7. Changing the parameters results in very interesting paths — feel free to explore the results with different m, k, and initial conditions!

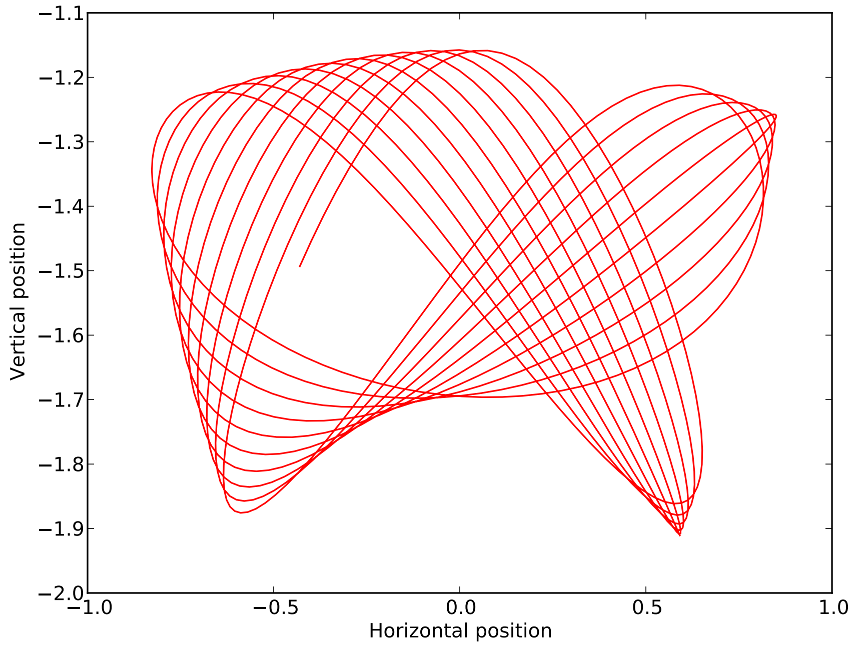

Figure 4.7: Path of the spring-pendulum in example 4.5.1, with initial conditions given in the program listing.

Let's look at one more example, this one with a time-dependent differential equation.

Example 4.5.2
Write a "derivs" function for a damped driven pendulum:

$$\ddot{\theta} = -\frac{g}{L}\sin\theta - b\dot{\theta} + \beta\cos\omega t$$

```
def pendulum_def(state, time):
    """
    The state variable should be defined in our
    usual way: y[0] = theta, y[1] = d(theta)/dt.
    The constants "gravity", "length", "beta",
    "b", and "omega" are assumed to be defined
    elsewhere.
    """
    g0 = y[1]
    g1 = -gravity/length * sin(y[0]) - b*y[1] +
        beta*cos(omega * time)
```

Incorporating time into our solving mechanism is easy, since we've included the *capacity* to use time since we started this "standard method".

4.6 Problems

4-0 Express each of these differential equations as a set of first-order differential equations

(a)
$$m\ddot{x} = f(x,t)$$

(b)
$$A\ddot{x} + B\dot{x} + Cx = D$$

(c)
$$m\ddot{\theta} = -\sqrt{\frac{g}{L}}\sin\theta - \beta\dot{\theta} + \gamma\sin\omega t$$

(d)
$$\dddot{y} + \ddot{y} + \dot{y} + y = 0$$

4-1 The number of radioactive atoms that decay in a given time period is proportional to the number of atoms in the sample:

$$\frac{dN}{dt} = -\lambda N$$

Write a program that uses Euler's method to plot $N(t)$. Have your program also plot the exact solution, $N(t) = N_o e^{-\lambda t}$, for comparison.

4-2 Write an appropriate "derivs" function for each of the differential equations in problem 0.

4-3 Write a function that impliments the 4^{th}-order Runge-Kutta method described in equation 4.22. Make sure it works by testing it for some previous problem or example that used Euler's method or second-order Runge-Kutta.

4-4 Write a function that solves ODEs for each point on an array of time values, such as is described on page 100. You may use whatever individual-step routine you wish (Euler, 4^{th}-order Runge-Kutta, etc.) Test your routine on a differential equation with a known solution, and show that it works.

4-5 In a radioactive decay chain, element A decays to element B which decays to C, and so on until the decay chain reaches a stable element. One example of such a chain is ^{90}Sr, which decays to ^{90}Y, which decays to ^{90}Zr. The half-life of ^{90}Sr is 28.78 years, and the half-life of ^{90}Y is 2.67 days. ^{90}Zr is stable. This decay chain can be described by the following differential equations:

$$\frac{dN_{Sr}}{dt} = -\lambda_{Sr} N_{Sr}$$

$$\frac{dN_Y}{dt} = -\lambda_Y N_Y - \frac{dN_{Sr}}{dt}$$

Plot the relative activity of a sample of ^{90}Sr as a function of time. (A logrithmic time scale will be helpful.)

4-6 A set of magnetically-coupled rotors (see [10]) with velocity-dependent damping move according to the equations

$$\ddot{\theta}_1 = \beta \sin(\theta_2 - \theta_1) - b\dot{\theta}_1$$
$$\ddot{\theta}_2 = \beta \sin(\theta_1 - \theta_2) - b\dot{\theta}_2$$

Write a program that plots θ_1 and θ_2 vs. time for a total time $\tau = 20$ seconds, with $b = 0.1$ and $\beta = 5.0$.

4-7 Redo problem 6 with damping that depends on $\dot{\theta}^2$ instead of on $\dot{\theta}$. Be careful about the sign of the damping term! You may also have to change the size of b to obtain interesting results.

4-8 A spring-mass system with friction, such as that in figure 4.4, is driven by an external force $F(t) = A\cos\omega t$.

a) Plot the frequency response of this system. In other words, for some constant value of A, what is the resulting amplitude of oscillation of the mass as a function of the driving frequency ω? Choose your parameters so as to span the "interesting" frequency region, and note that a logrithmic graph axis might be helpful on one or both axes.

b) How does the frequency response change with driving amplitude A? Write a program to make a graph that answers the question.

It is possible to write a single program that answers both parts of this question with one graph, although you may find that 2-dimensional graphs are somewhat limiting for this purpose.

4-9 The rate at which a finite resource (such as oil) is gathered depends on the difficulty and on the demand. Assume that the demand D increases with the amount extracted E:

$$\frac{d}{dt}[D] = \alpha E$$

Assume that the difficulty of extraction W is inversely dependent on the fraction remaining:

$$W = \frac{1}{1-E}$$

Finally, assume that the rate of extraction R is the ratio of demand to difficulty:

$$R = \frac{d}{dt}[E] = \frac{D}{W}$$

Assume also that the cost C of the resource depends on the extraction rate and on the demand:

$$C = \frac{D}{R}\xi$$

where ξ is some scale factor.

Write a program that plots the extraction rate and the cost as a function of time, for this simple model. Good starting values are $D_o = 0.1$, $\alpha = 2.0$, and $\xi = 0.01$, plotted over a total time range of 0 to 5.