ODEs

Carver J. Bierson

July 18, 2017

Almost all of the most descriptive equations are differential equations. In many cases, as is a recurring theme, these equations can not be solved analytically either because of the form of the equations or their boundary conditions. We will start by solving a first order differential equation, i.e.,

$$\frac{df(t)}{dt} = g(t, f) \tag{1}$$

where g can be any function of f and t. A more concrete example is

$$\frac{df(t)}{dt} = t^2 \tag{2}$$

When we have a differential equation like those above we want to solve for is $f(t)$, given some initial state. To do this we will take the equation and put it in terms we can evaluate numerically. Conceptually derivatives describe the slope of a curve or equivalently, how much $f(t)$ changes per a change in t. Given this we will approximate $df(t)/dt$ as $\Delta f(t)/\Delta t$. Now we can rearrange our equation to be

$$\Delta f(t) = t^2 \Delta t \tag{3}$$

If we know $f(0)$ we can walk our way through time in steps of Δt, updated our knowledge of $f(t)$ as we go. We can write this out is stensil form as

$$f(t_{i+1}) - f(t_i) = t_i^2 \Delta t \tag{4}$$

where the i subscript indicates which step in this process we are on. Here is what that looks like in python assuming $f(0) = 5$. In this case we also know the analytic solution so I will plot that as well

```
import matplotlib.pyplot as plt
import numpy as np

#f'(t)=t^2
def dfdt(t):
    return t**2

dt=1.5 # Our time step (s)

t=0 # our initial time
t_max=20 # The time we want to find our solution at

f=np.array([5]) # f(0)=5
i=0

```

```
while t<=t_max:
    i+=1# index where we are in our array

    # make the array f larger by one value to store our new value
    f=np.append(f,0.0)
    # calculate our new f(t+dt) value
    f[i]=dfdt(t)*dt+f[i-1]
    # increment time
    t+=dt

# create a time array to plot against
time_arr=np.arange(0,t+dt,dt)

# Plot result
plt.figure()

plt.plot(time_arr, 5.0+1/3.0*time_arr**3.0,
      label='Analytic Solution')
plt.plot(time_arr,f,'r-*', label='Numerical Solution')

plt.xlabel('t')
plt.ylabel('f(t)')
plt.legend()

plt.show()
```

You will notice in the plot that our numerical solution underestimates the true solution. The smaller you make dt, smaller the error in our solution will be.

The above method is known as a forward Euler solver. This is the simplest ODE solver and is good enough for many applications, but it is far from the best.

1 Higher order derivatives

Lets consider the classics second order derivative for the position of a mass on a spring.

$$\frac{d^2x}{dt^2} = -\frac{k}{m}x \tag{5}$$

We can write this as two ordinary differential equations via

$$\frac{dx}{dt} = v \tag{6}$$

$$\frac{dv}{dt} = -\frac{k}{m}x \tag{7}$$

Note that in this cases as long as we know the initial state (position and velocity) we can solve for the state at any future time. This set is known a set of coupled ODEs because their equations depend on one another. Solving sets of coupled ODEs allows us to solve many complex systems including multiple springs connected in series.

2 Using the scipy integrator

Scipy contains a general ode integrator called *odeint*.

```python
import matplotlib.pyplot as plt
import numpy as np

import scipy.integrate as integ

# define our differential equation
def spring_derivs(xv, t, k, m):
    x = xv[0]
    v = xv[1]
    dxdt = v
    dvdt = -(k/m)*x
    return( [dxdt, dvdt] )


# define constants
m=1.0 # km
k=2.0 # N/m

# define initial state
xv0=[1,0] # x=1 m, v=0 m/s

# Make an array of times to solve at
t=np.linspace(0,20,100)

# now we can pass this to odeint via
output = integ.odeint(spring_derivs, xv0, t, args=(k, m))

#output now contains the solved values for x and v at each t in
    each column
x=output[:,0]
v=output[:,1]

# plot the results
plt.figure()

plt.plot(t, x,label='x (m)')
plt.plot(t, v, label='v (m/s)')

plt.xlabel('t')
plt.legend()

plt.show()
```

3 In Class Problems

- Compare the odeint result to the analytic result for the integral of t^2.
- Trying including a damping force ($f_d = -c_v v$) to the spring system and compare the output.
- In the case where there is damping, eventually the spring comes to rest. Compute the total distance traveled by the spring in the limit of time going to infinity. In practice, you can truncate the integration after a very long time. Hint: total distance traveled is the integral of the absolute value of the velocity, so take your velocity output by the ODE solver, take its absolute value, and then integrate.
- Choose values of k and m, and then repeat part the previous problem for a range of different values of c_v. Make a plot of distance traveled versus

c_v. You should find that the function decreases from infinity at $c_v = 0$ to 1 as c_v approaches infinity, but that there is a special value of c_v above which the distance traveled is about constant. What is this value of c_v, and what is its physical significance?