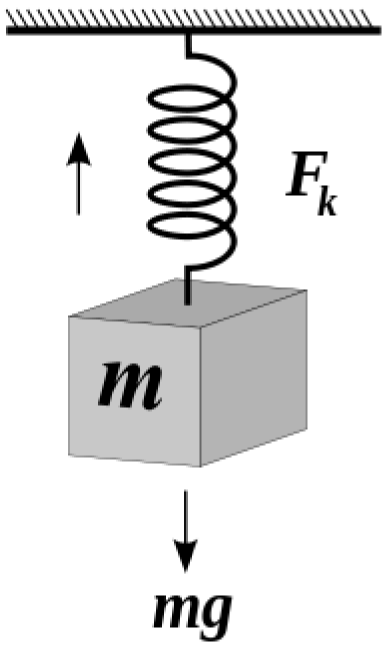
# Modelling Dynamical Systems

## Characterizing a System Using Differential Equations

A dynamical system such as the mass-spring system we saw before, can be characterized by the relationship between state variables $s$ and their (time) derivatives $\dot{s}$. How do we arrive at the correct characterization of this relationship? The short answer is, we figure it out using our knowledge of physics, or we are simply given the equations by someone else. Let's look at a simple mass-spring system again.



A spring with a mass attached

We know a couple of things about this system. We know from Hooke's law of elasticity that the extension of a spring is directly and linearly proportional to the load applied to it. More precisely, the force that a spring applies in response to a perturbation from it's *resting length* (the length at which it doesn't generate any force), is linearly proportional, through a constant $k$, to the difference in length between its current length and its resting length (let's call this distance $x$). For convention let's assume positive values of $x$ correspond to lengthening the spring beyond its resting length, and negative values of $x$ correspond to shortening the spring from its resting length.

$F = -kx$

Let's decide that the *state variable* that we are interested in for our system is $x$. We will refer to $x$ instead of $s$ from now on to denote our state variable.

We also know from Newton's laws of motion (specifically Newton's second law) that the net force on an object is equal to its mass $m$ multiplied by its acceleration $a$ (the second derivative of position).

$F = ma$

Instead of using $a$ to denote acceleration let's use a different notation, in terms of the spring's perturbed length $x$. The rate of change (velocity) is denoted $\dot{x}$ and the rate of change of the velocity (i.e. the acceleration) is denoted $\ddot{x}$.

$F = m\ddot{x}$

We also know that the mass is affected by two forces: the force due to the spring ($-kx$) and also the gravitational force $g$. So the equation characterizing the *net forces* on the mass is

$\Sigma F = m\ddot{x} = -kx + mg$

or just

$m\ddot{x} = -kx + mg$

This equation is a *second-order* differential equation, because the highest state derivative is a *second derivative* (i.e. $\ddot{x}$, the second derivative, i.e. the acceleration, of $x$). The equation specifies the relationship between the state variables (in this case a single state variable $x$) and its derivatives (in this case a single derivative, $\ddot{x}$).

The reason we want an equation like this, from a practical point of view, is that we will be using numerical solvers in Python/Scipy to *integrate* this differential equation over time, so that we can *simulate* the behaviour of the system. What these solvers need is a Python function that returns state derivatives, given current states. We can re-arrange the equation above so that it specifies how to compute the state derivative $\ddot{x}$ given the current state $\ddot{x}$.

$\ddot{x} = -kxm + g$

Now we have what we need in order to simulate this system in Python/Scipy. At any time point, we can compute the acceleration of the mass by the formula above.

## Integrating Differential Equations in Python/SciPy

Here is a Python function that we will be using to simulate the mass-spring system. All it does, really, is compute the equation above: what is the value of $\ddot{x}$, given $x$? The one addition we have is that we are going to keep track not just of one state variable $x$ but also its first derivative $\dot{x}$ (the rate of change of $x$, i.e. velocity).

```
def MassSpring(state,t):
  # unpack the state vector
  x = state[0]
  xd = state[1]

  # these are our constants
  k = 2.5 # Newtons per metre
  m = 1.5 # Kilograms
  g = 9.8 # metres per second

  # compute acceleration xdd
  xdd = ((-k*x)/m) + g

  # return the two state derivatives
  return [xd, xdd]
```

Note that the function we wrote takes two arguments as inputs: state and t, which corresponds to time. This is necessary for the numerical solver that we will use in Python/Scipy. The state variable is actually an *array* of two values corresponding to $x$ and $\dot{x}$.

How does numerical integration (simulation) work? Here is a summary of the steps that a numerical solver takes. First, you have to provide the system with two things:

1. initial conditions (what are the initial states of the system?)
2. a time vector over which to simulate

Given this, the numerical solver will go through the following steps to simulate the system:

- calculate state derivatives $\ddot{x}$ at the initial time ($t=0$) given the initial states ($x, \dot{x}$)
- estimate $x(t+\Delta t)$ using $x(t=0)$, $\dot{x}(t=0)$ and $\ddot{x}(t=0)$
- calculate $\ddot{x}(t=t+\Delta t)$ from $x(t=t+\Delta t)$ and $\dot{x}(t=t+\Delta t)$
- estimate $x(t+2\Delta t)$ and $\dot{x}(t+2\Delta t)$ using $x(t=t+\Delta t)$, $\dot{x}(t=t+\Delta t)$ and $\ddot{x}(t=t+\Delta t)$
- calculate $\ddot{x}(t=t+2\Delta t)$ from $x(t=t+2\Delta t)$ and $\dot{x}(t=t+2\Delta t)$
- … etc

In this way the numerical solver can esimate how the system states ($x, \dot{x}$) unfold over time, given the initial conditions, and the known relationship between state derivatives and system states. The details of the "estimate" steps above are not something we are going to dive into now. Suffice it to say that current estimation algorithms are based on the work of two German mathematicians named Runge and Kutta in the beginning of the 20th century. These numerical recipies are readily available in Scipy (docs here(and in MATLAB, and other numerical software) and are known as ODE solvers (ODE stands for *ordinary differential equation*).

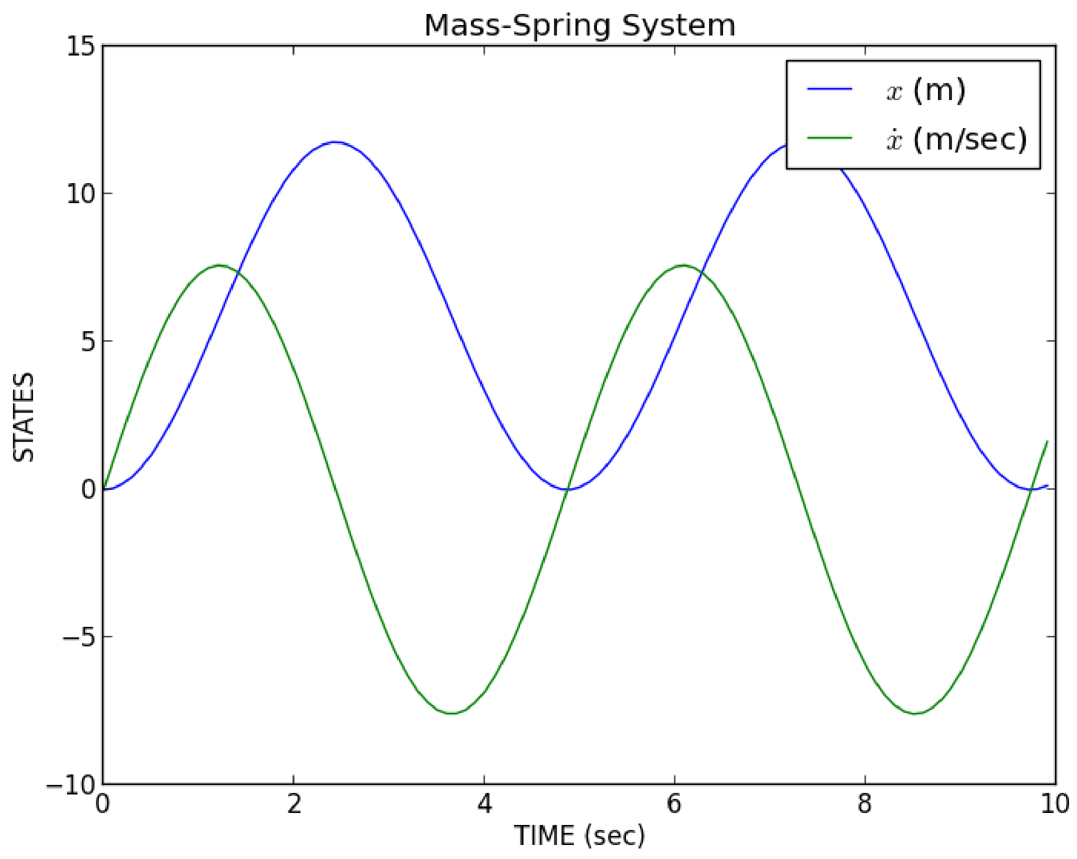
Here's how we would simulate the mass-spring system above. Launch iPython with the --pylab argument (this automatically imports a bunch of libraries that we will use, including plotting libraries).

```
from scipy.integrate import odeint

def MassSpring(state,t):
  # unpack the state vector
  x = state[0]
  xd = state[1]

  # these are our constants
  k = -2.5 # Newtons per metre
  m = 1.5 # Kilograms
  g = 9.8 # metres per second

  # compute acceleration xdd
  xdd = ((k*x)/m) + g
  # return the two state derivatives
  return [xd, xdd]

state0 = [0.0, 0.0]
t = arange(0.0, 10.0, 0.1)

state = odeint(MassSpring, state0, t)

plot(t, state)
xlabel('TIME (sec)')
ylabel('STATES')
title('Mass-Spring System')
legend(('$x$ (m)', '$\dot{x}$ (m/sec)'))
```

mass_spring.py

A couple of notes about the code. I have simply chosen, out of the blue, values for the constants *k* and *m*. The gravitational constant *g* is of course known. I have also chosen to simulate the system for 10 seconds, and I have chosen a time *resolution* of 100 milliseconds (0.1 seconds). We will talk later about the issue of what is an appropriate time resolution for simulation.

You should see a plot like this:



Mass-Spring Simulation

The blue line shows the position *x* of the mass (the length of the spring) over time, and the green line shows the rate of change of *x*, in other words the velocity $\dot{x}$, over time. These are the two states of the system, simulated over time.

The way to interpret this simulation is, if we start the system at $x=0$ and $\dot{x}=0$, and simulate for 10 seconds, this is how the system would behave.

The power of modelling and simulation

Now you can appreciate the power of mathematical models and simulation: given a model that characterizes (to some degree of accuracy) the behaviour of a system we are interested in, we can use simulation to perform experiments *in simulation* instead of in reality. This can be very powerful. We can ask questions of the model, in simulation, that may be too difficult, or expensive, or time consuming, or just plain impossible, to do in real-life empirical studies. The degree to which we regard the results of simulations as interpretable, is a direct reflection of the degree to which we believe that our mathematical model is a reasonable characterization of the behaviour of the real system.

## Exercises

1. We have started the system at $x=0$ which means that the spring is not stretched beyond its resting length (so spring force due to stretch should equal zero), and $\dot{x}=0$, which means the spring's velocity is zero, i.e. it is not moving. Why does the simulation predict that the spring will begin stretching, then bouncing back and forth?
2. What is the influence of the sign and magnitude of the stiffness parameter $k$?
3. In physics, damping can be used to reduce the magnitude of oscillations. Damping generates a force that is directly proportional to velocity ($F=-b\dot{x}$). Add damping to the mass-spring system and re-run the simulation. Specify the value of the damping constant $b=-2.0$. What happens?
4. What is the influence of the sign and magnitude of the damping coefficient $b$?
5. Double the mass, and re-run the simulaton. What happens?
6. How would you add an input force to the system?

## Lorenz Attractor

The Lorenz system is a dynamical system that we will look at briefly, as it will allow us to discuss several interesting issues around dynamical systems. It is a system often used to illustrate non-linear systems theory and chaos theory. It's sometimes used as a simple demonstration of the butterfly effect (sensitivity to initial conditions).

The Lorenz system is a simplified mathematical model for atmospheric convection. Let's not worry about the details of what it represents, for now the important things to note are that it is a system of three *coupled* differential equations, and characterizes a system with three state variables $(x,y,z)$.

$$\dot{x}\,\dot{y}\,\dot{z}===\sigma(y-x)(\rho-z)x-yxy-\beta z$$

If you set the three constants $(\sigma,\rho,\beta)$ to specific values, the system exhibits *chaotic behaviour*.

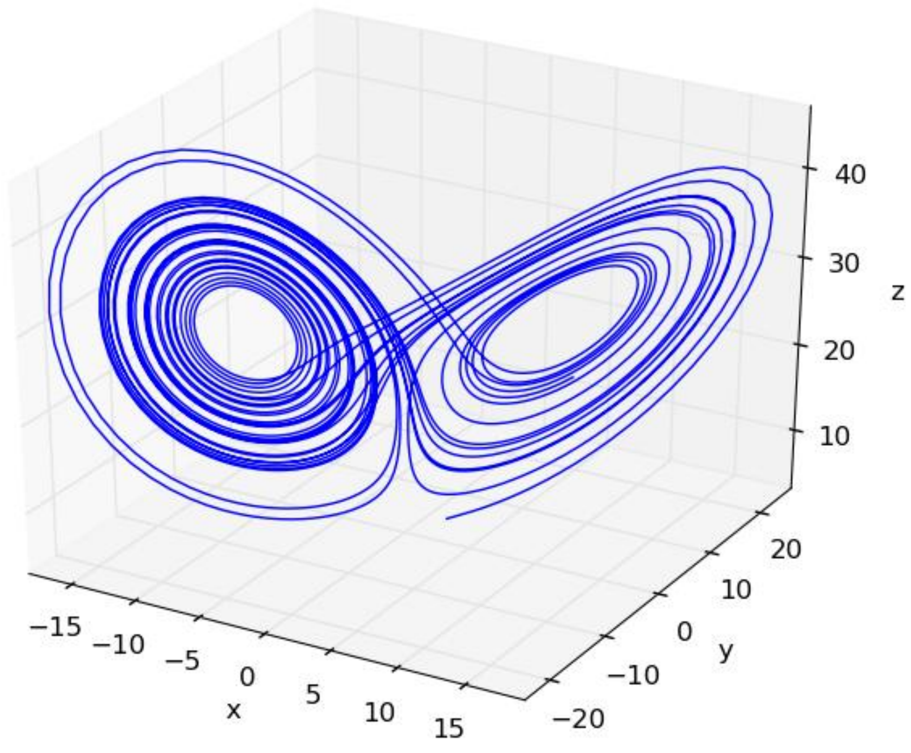
$$\sigma\rho\beta===102883$$

Let's implement this system in Python/Scipy. We have been given above the three equations that characterize how the state derivatives ($\dot{x}$, $\dot{y}$, $\dot{z}$) depend on ($x$,$y$,$z$) and the constants ($\sigma$,$\rho$,$\beta$). All we have to do is write a function that implements this, set some initial conditions, decide on a time array to simulate over, and run the simulation using odeint().

```
from scipy.integrate import odeint

def Lorenz(state,t):
  # unpack the state vector
  x = state[0]
  y = state[1]
  z = state[2]

  # these are our constants
  sigma = 10.0
  rho = 28.0
  beta = 8.0/3.0

  # compute state derivatives
  xd = sigma * (y-x)
  yd = (rho-z)*x - y
  zd = x*y - beta*z

  # return the state derivatives
  return [xd, yd, zd]

state0 = [2.0, 3.0, 4.0]
t = arange(0.0, 30.0, 0.01)

state = odeint(Lorenz, state0, t)

# do some fancy 3D plotting
from mpl_toolkits.mplot3d import Axes3D
fig = figure()
ax = fig.gca(projection='3d')
ax.plot(state[:,0],state[:,1],state[:,2])
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
show()
```

lorenz1.py

You should see something like this:



Lorenz Attractor

The three axes on the plot represent the three states $(x,y,z)$ plotted over the 30 seconds of simulated time. We started the system with three particular values of $(x,y,z)$(I chose them arbitrarily), and we set the simulation in motion. This is the trajectory, in *state-space*, of the Lorenz system.

You can see an interesting thing... the system seems to have two stable equilibrium states, or attractors: those circular paths. The system circles around in one "neighborhood" in state-space, and then flips over and circles around the second neighborhood. The number of times it circles in a given neighborhood, and the time at which it switches, displays chaotic behaviour, in the sense that they are exquisitly sensitive to initial conditions.

For example let's re-run the simulation but change the initial conditions. Let's change them by a very small amount, say 0.0001... and let's only change the $x$ initial state by that very small amount. We will simulate for 30 seconds.

```
t = arange(0.0, 30, 0.01)

# original initial conditions
state1_0 = [2.0, 3.0, 4.0]
state1 = odeint(Lorenz, state1_0, t)

# rerun with very small change in initial conditions
delta = 0.0001
state2_0 = [2.0+delta, 3.0, 4.0]
state2 = odeint(Lorenz, state2_0, t)

# animation
figure()
pb, = plot(state1[:,0],state1[:,1],'b-',alpha=0.2)
xlabel('x')
ylabel('y')
p, = plot(state1[0:10,0],state1[0:10,1],'b-')
pp, = plot(state1[10,0],state1[10,1],'b.',markersize=10)
p2, = plot(state2[0:10,0],state2[0:10,1],'r-')
pp2, = plot(state2[10,0],state2[10,1],'r.',markersize=10)
tt = title("%4.2f sec" % 0.00)
# animate
```

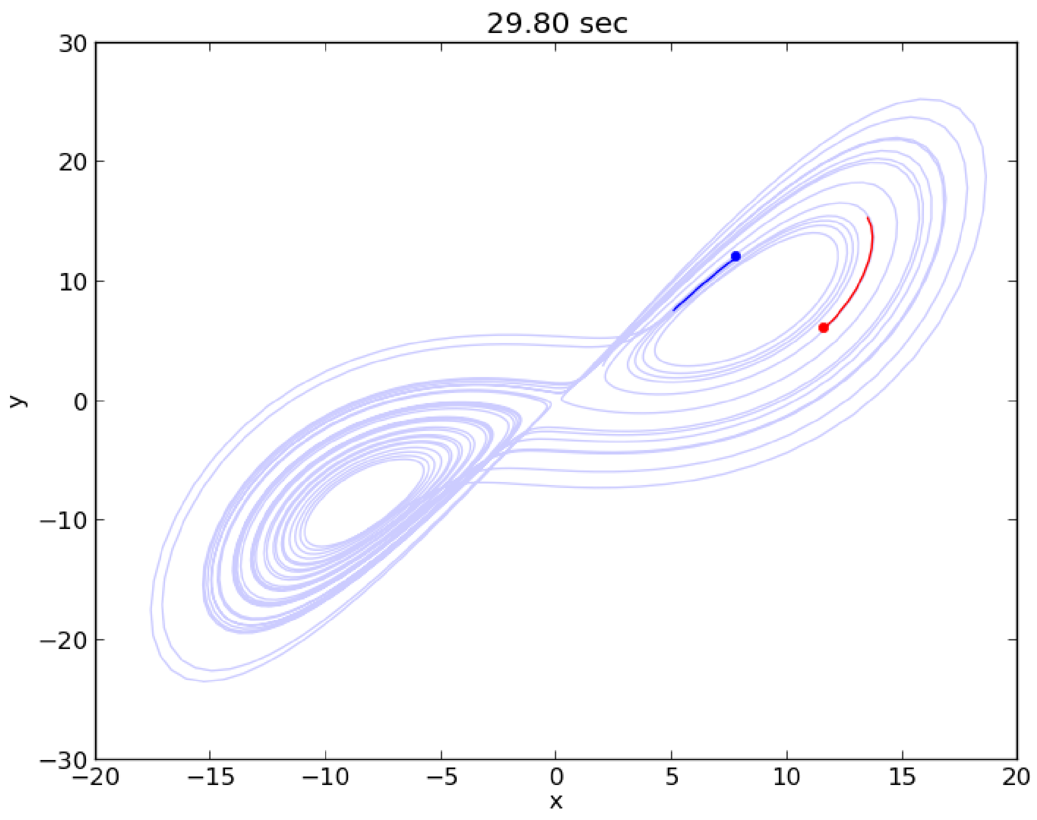
```
step = 3
for i in xrange(1,shape(state1)[0]-10,step):
  p.set_xdata(state1[10+i:20+i,0])
  p.set_ydata(state1[10+i:20+i,1])
  pp.set_xdata(state1[19+i,0])
  pp.set_ydata(state1[19+i,1])
  p2.set_xdata(state2[10+i:20+i,0])
  p2.set_ydata(state2[10+i:20+i,1])
  pp2.set_xdata(state2[19+i,0])
  pp2.set_ydata(state2[19+i,1])
  tt.set_text("%4.2f sec" % (i*0.01))
  draw()

i = 1939          # the two simulations really diverge here!
s1 = state1[i,:]
s2 = state2[i,:]
d12 = norm(s1-s2) # distance
print ("distance = %f for a %f different in initial condition") % (d12, delta)
```

lorenz2.py

```
distance = 32.757253 for a 0.000100 different in initial condition
```

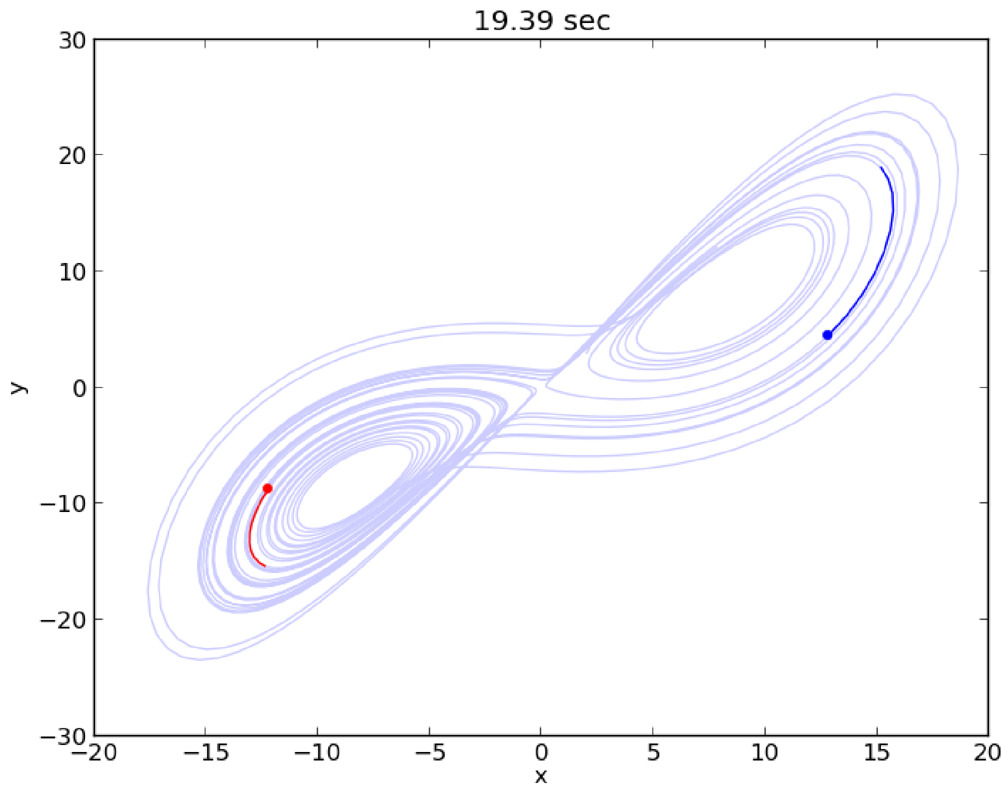
You should see an animation of the two state-space trajectories. For convenience we are only plotting *x* vs *y* and ignoring *z*. It turns out that 3D animations are not trivial in matplotlib (there is a library called mayavi that is excellent for 3D stuff).

The original simulation is shown in blue and the new one (in which the initial condition of *x* was increased by 0.0001) in red. The two follow each other quite closely for a long time, and then begin to diverge at about the 16 second mark. At the end of the animation it looks like this:



Lorenz Attractor

At 19.39 seconds it looks like this:



Lorenz Attractor

Note how the two systems are in different "neighborhoods" entirely!

At the end of the code above we compute the distance between the two systems (the 3D distance between their respective $(x,y,z)$ positions in state-space), and the distance is a whopping 32.76 units, for a 0.0001 difference in initial conditions.

This illustrates how systems with relatively simple differential equations characterizing their behaviour, can turn out to be exquisitely sensitive to initial conditions. Just imagine if the initial conditions of your simulation were gathered from empirical observations (like the weather, for example). Now imagine you use a model simulation to predict whether it will be sunny (left-hand "neighborhood" of the plot above) or thunderstorms (right-hand "neighborhood"), 30 days from now. If the answer can flip between one prediction and the other, based on a 1/10,000 different in measurement, you had better be sure of your empirical measurement instruments, when you make a prediction 30 days out! Actually this won't even solve the problem, no matter how precise your measurements. The point is that the system as a whole is very sensitive to even tiny changes in initial conditions. This is why short-term weather forecasts are relatively accurate, but forecasts past a couple of days can turn out to be dead wrong.

Predator-Prey model

The Lotka-Volterra equations are two coupled first-order nonlinear differential equations that are used to characterize the dynamics of biological systems in which a predator population and a prey popuation interact. The two populations develop over time according to these equations:

$$\dot{x}\ \dot{y} == x(\alpha-\beta y) - y(\gamma-\sigma x)$$

where $x$ is the number of prey (for example, rabbits), $y$ is the number of predators (e.g. foxes), and $\dot{x}$ and $\dot{y}$ represent the growth rates (the rates of change over time) of the two populations. The values $(\alpha,\beta,\gamma,\sigma)$ are parameters (constants) that characterize different aspects of the two populations.

Assumptions of this simple form of the model are:

1. prey find ample food at all times
2. food supply of predators depends entirely on prey population
3. rate of change of population is proportional to its size
4. the environment does not change

The parameters can be interepreted as:

- $\alpha$ is the natural growth rate of prey in the absence of predation
- $\beta$ is the death rate per encounter of prey due to predation
- $\sigma$ is related to the growth rate of predators
- $\gamma$ is the natural death rate of predators in the absence of food (prey)

Here is some example code showing how to simulate this system. Just as before, we need to complete a few steps:

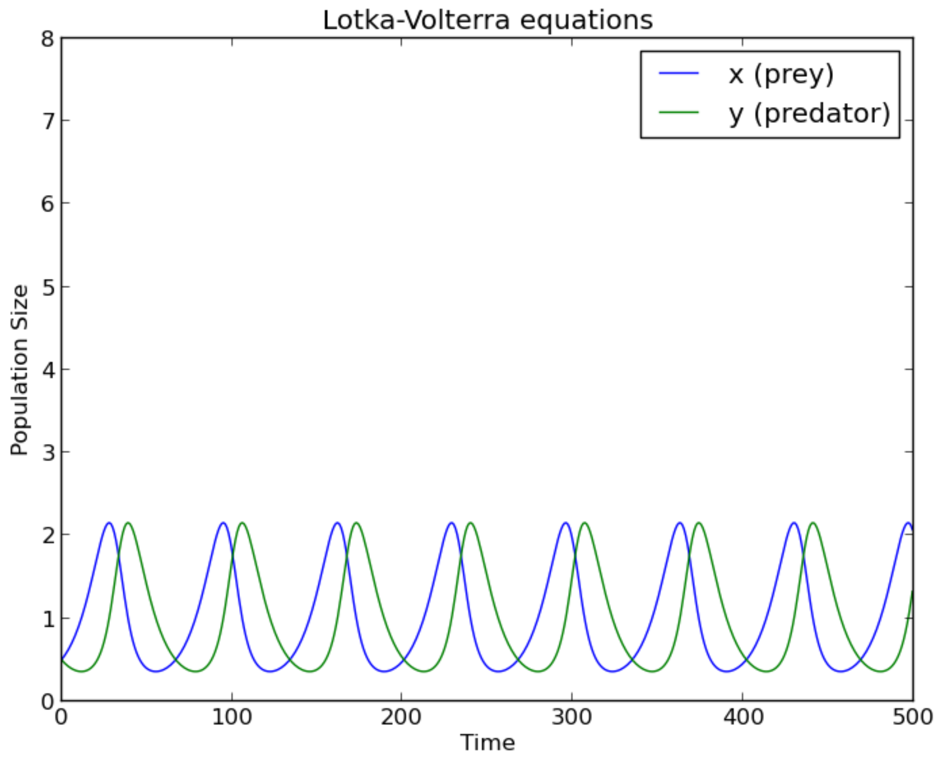
1. write a Python function that characterizes how the system's state derivatives are related to the system's states (this is given by the equations above)
2. decide on values of the system parameters
3. decide on values of the initial conditions of the system (the initial values of the states)
4. decide on a time span and time resolution for simulating the system

5. simulate! (i.e. use an ODE solver to integrate the differential equations over time)
6. examine the states, typically by plotting them

Here is some code:

```
from scipy.integrate import odeint

def LotkaVolterra(state,t):
  x = state[0]
  y = state[1]
  alpha = 0.1
  beta =  0.1
  sigma = 0.1
  gamma = 0.1
  xd = x*(alpha - beta*y)
  yd = -y*(gamma - sigma*x)
  return [xd,yd]

t = arange(0,500,1)
state0 = [0.5,0.5]
state = odeint(LotkaVolterra,state0,t)
figure()
plot(t,state)
ylim([0,8])
xlabel('Time')
ylabel('Population Size')
legend(('x (prey)','y (predator)'))
title('Lotka-Volterra equations')
```

You should see a plot like this:



Lotka-Volterra Simulation

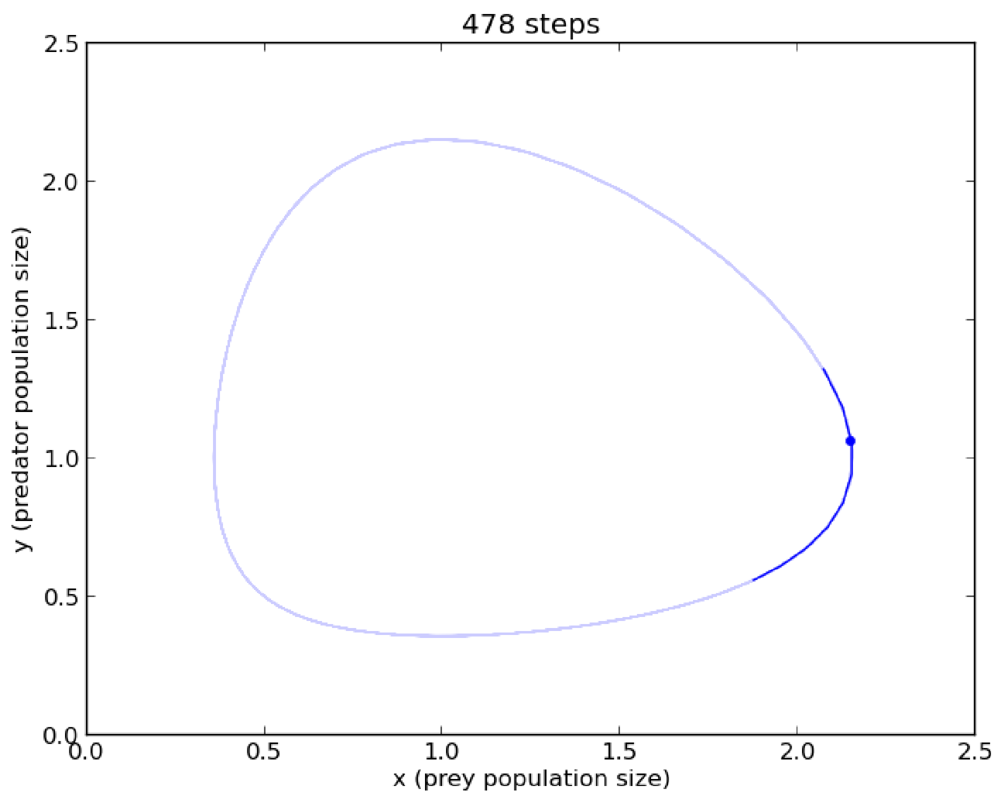
We can also plot the trajectory of the system in *state-space* (much like we did for the Lorenz system above):

```
# animation in state-space
figure()
pb, = plot(state[:,0],state[:,1],'b-',alpha=0.2)
xlabel('x (prey population size)')
ylabel('y (predator population size)')
p, = plot(state[0:10,0],state[0:10,1],'b-')
pp, = plot(state[10,0],state[10,1],'b.',markersize=10)
tt = title("%4.2f sec" % 0.00)

# animate
step=2
for i in xrange(1,shape(state)[0]-10,step):
  p.set_xdata(state[10+i:20+i,0])
  p.set_ydata(state[10+i:20+i,1])
```

```
pp.set_xdata(state[19+i,0])
pp.set_ydata(state[19+i,1])
tt.set_text("%d steps" % (i))
draw()
```

lotkavolterra.py

You should see a plot like this:



Lotka-Volterra State-space plot

Source:
http://www.gribblelab.org/compneuro2012/2_Modelling_Dynamical_Systems.html