Differential Equations

Pretty common in physics! This should look pretty common

$$\ddot{x}(t) = -g$$

$$\dot{x}(t) = -gt + v_0 t$$

$$x(t) = -\frac{1}{2} g t^2 + v_0 t + x_0$$

But what if we add some other, arbitrary forces? We can't solve analytically like above

$$\ddot{x}(t) = F(x, t)$$

First-order ordinary differential equations

We'll start with these sorts of equations

$$\frac{dx}{dt} = \dot{x} = f(x,t)$$

We can do a simple Taylor explanation,
giving us Euler's method:

$$x(t+h) = x(t) + h\frac{dx}{dt} + \frac{1}{2}h^2\frac{d^2x}{dt^2} + \ldots$$

$$x(t+h) = x(t) + hf(x,t) + \mathcal{O}(h^2) + \ldots$$

What is Euler's method doing?

$f_n = f(t_n)$, $t_n = nh$ for time slice size h

The method updates the "velocity" and "acceleration" after each step

$$v_{n+1} = v_n + h a_n$$
$$x_{n+1} = x_n + h v_n$$

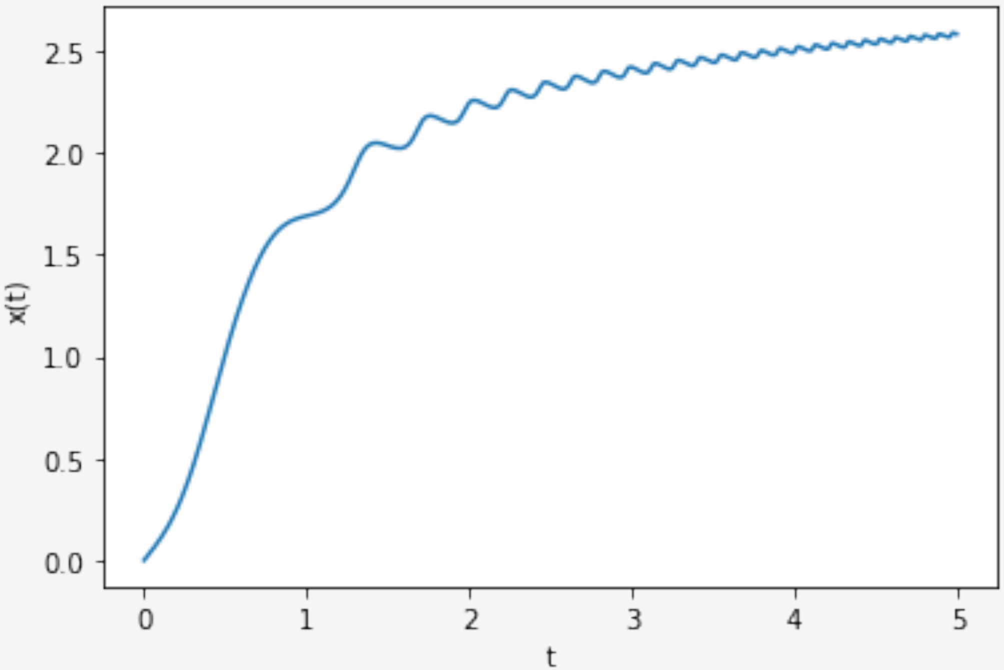
$$x(t+h) = x(t) + hf(x,t) + \mathcal{O}(h^2) + \ldots$$

Each step has size h, and each step has error of size h^2. In total time T we take N steps = T/h, so at the Nth step the error is then $(T/h)*h^2$, ie O(h), not great

First-order ordinary differential equations

Euler's method:

```
[1] ### Euler's method to solve dx/dt = e^(-x^2 + 3)*t^2 + cos(x^2*t^2) from t = 0 to t = 5 with x(0) = 0
    from math import cos,exp
    from numpy import arange
    from pylab import plot,xlabel,ylabel,show

    def f(x,t):
      return exp(-x**2+3)*t**2+cos(x**2*t**2)

    a = 0.0 ## t=0 start
    b = 5.0 ## max t
    N = 1000 ## number of steps
    h = (b-a)/N ## size of single step
    x = 0.0 ## initial position

    tpoints = arange(a,b,h)
    xpoints = []
    for t in tpoints:
      xpoints.append(x)
      x += h*f(x,t)

    plot(tpoints,xpoints)
    xlabel("t")
    ylabel("x(t)")
    show()
```



Second-order Runge-Kutta method

Before, we found x(t+h) by expanding around x(t):

$$x(t+h) = x(t) + h\frac{dx}{dt} + \frac{1}{2}h^2\frac{d^2x}{dt^2} + \ldots$$

Let's try expanding around x(t+h/2) instead:

$$x(t+h) = x(t+h/2) + (h/2)\left(\frac{dx}{dt}\right)_{t+h/2} + \frac{1}{8}h^2\left(\frac{d^2x}{dt^2}\right)_{t+h/2} + \mathcal{O}(h^3)$$

$$x(t) = x(t+h/2) - (h/2)\left(\frac{dx}{dt}\right)_{t+h/2} + \frac{1}{8}h^2\left(\frac{d^2x}{dt^2}\right)_{t+h/2} + \mathcal{O}(h^3)$$

Second-order Runge-Kutta method

$$x(t+h) = x(t+h/2) + (h/2)\left(\frac{dx}{dt}\right)_{t+h/2} + \frac{1}{8}h^2\left(\frac{d^2x}{dt^2}\right)_{t+h/2} + \mathcal{O}(h^3)$$

$$x(t) = x(t+h/2) - (h/2)\left(\frac{dx}{dt}\right)_{t+h/2} + \frac{1}{8}h^2\left(\frac{d^2x}{dt^2}\right)_{t+h/2} + \mathcal{O}(h^3)$$

Combine:

$$x(t+h) = x(t) + h\left(\frac{dx}{dt}\right)_{t+h/2} + \mathcal{O}(h^3)$$

$$x(t+h) = x(t) + hf(x(t+h/2), t+h/2) + \mathcal{O}(h^3)$$

What is x(t+h/2)??? Use Euler's method!

Second-order Runge-Kutta method

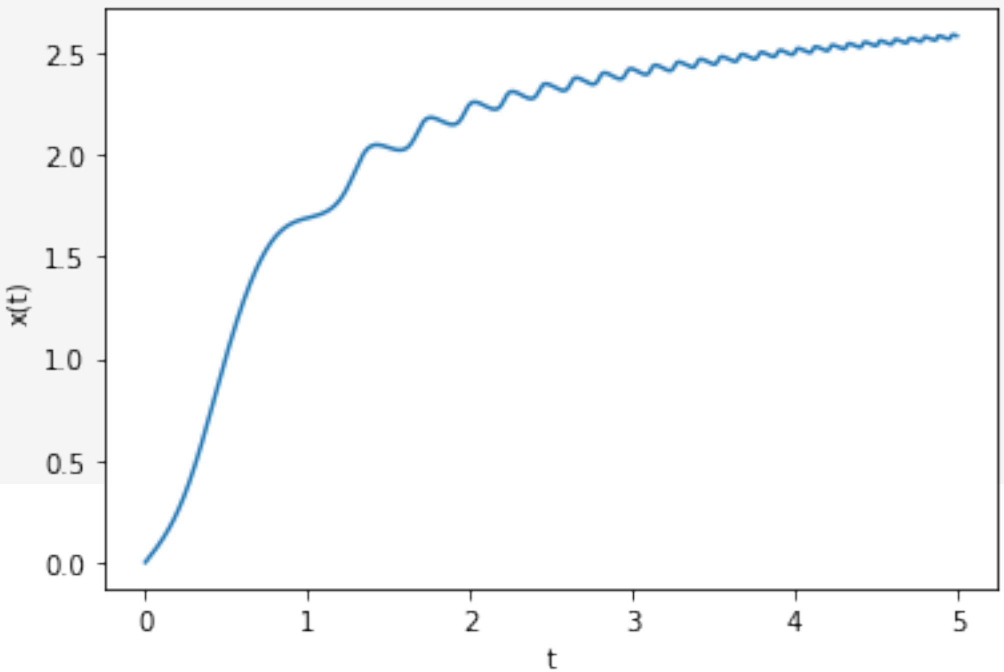
$$x(t+h) = x(t) + hf(x(t+h/2), t+h/2) + \mathcal{O}(h^3)$$

$$x(t+h/2) = x(t) + \frac{h}{2} f(x,t)$$

So we first find x(t+h/2) and then use that to find x(t+h)

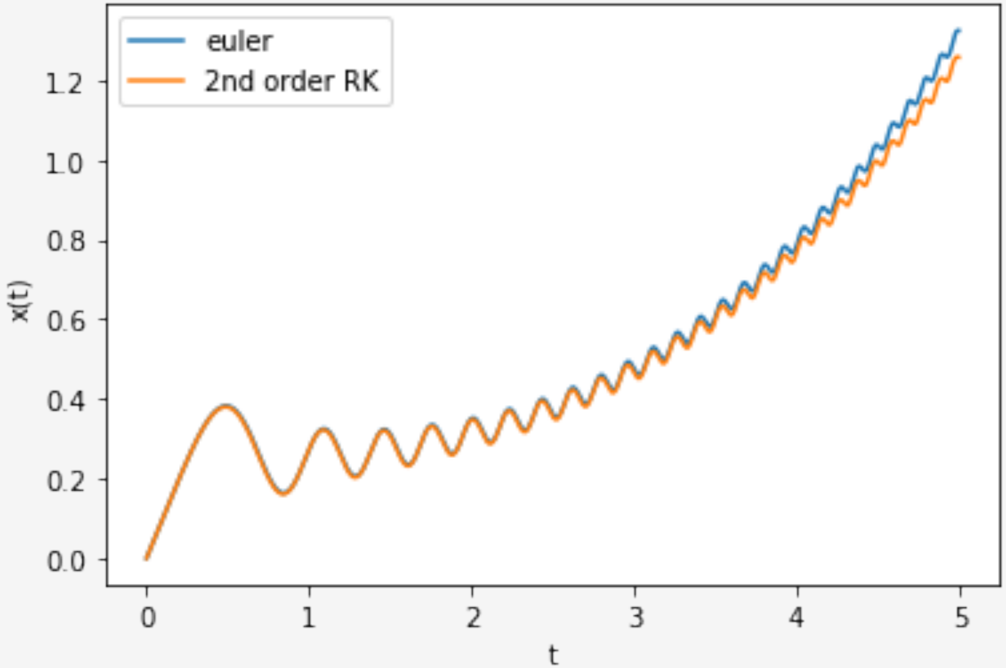
Second-order Runge-Kutta method


```python
### Second-order Runge Kutta method to solve dx/dt = e^(-x^2 + 3)*t^2 + cos(x^2*t^2) from t = 0 to t = 5 with x(0) = 0
from math import cos,exp
from numpy import arange
from pylab import plot,xlabel,ylabel,show

def f(x,t):
  return exp(-x**2+3)*t**2+cos(x**2*t**2)

a = 0.0 ## t=0 start
b = 5.0 ## max t
N = 1000 ## number of steps
h = (b-a)/N ## size of single step
x = 0.0 ## initial position

tpoints = arange(a,b,h)
xpoints = []
for t in tpoints:
  xpoints.append(x)
  k1 = h*f(x,t)
  k2 = h*f(x+0.5*k1,t+0.5*h)
  x += k2

plot(tpoints,xpoints)
xlabel("t")
ylabel("x(t)")
show()
```



Second-order Runge-Kutta method, closer look


```
### Second-order Runge Kutta method vs Euler to solve dx/dt = e^(-x^2 + 3)*t^2 + cos(x^2*t^2) from t = 0 to t = 5 with x(0) = 0
### Smaller number of steps!
from math import cos,exp
from numpy import arange
from pylab import plot,xlabel,ylabel,show,legend

def f(x,t):
  return exp(-x**2+3)*t**2+cos(x**2*t**2)

a = 0.0 ## t=0 start
b = 5.0 ## max t
N = 1000 ## number of steps
h = (b-a)/N ## size of single step
x_euler = 0.0 ## initial position
x_2rk = 0.0 ## initial position

tpoints = arange(a,b,h)
xpoints_2rk = []
xpoints_euler = []
for t in tpoints:
  xpoints_2rk.append(x_2rk)
  xpoints_euler.append(x_euler)
  k1 = h*f(x,t)
  k2 = h*f(x+0.5*k1,t+0.5*h)
  x_2rk += k2
  x_euler += h*f(x,t)

plot(tpoints,xpoints_euler,label="euler")
plot(tpoints,xpoints_2rk,label="2nd order RK")
xlabel("t")
ylabel("x(t)")
legend()
show()
```

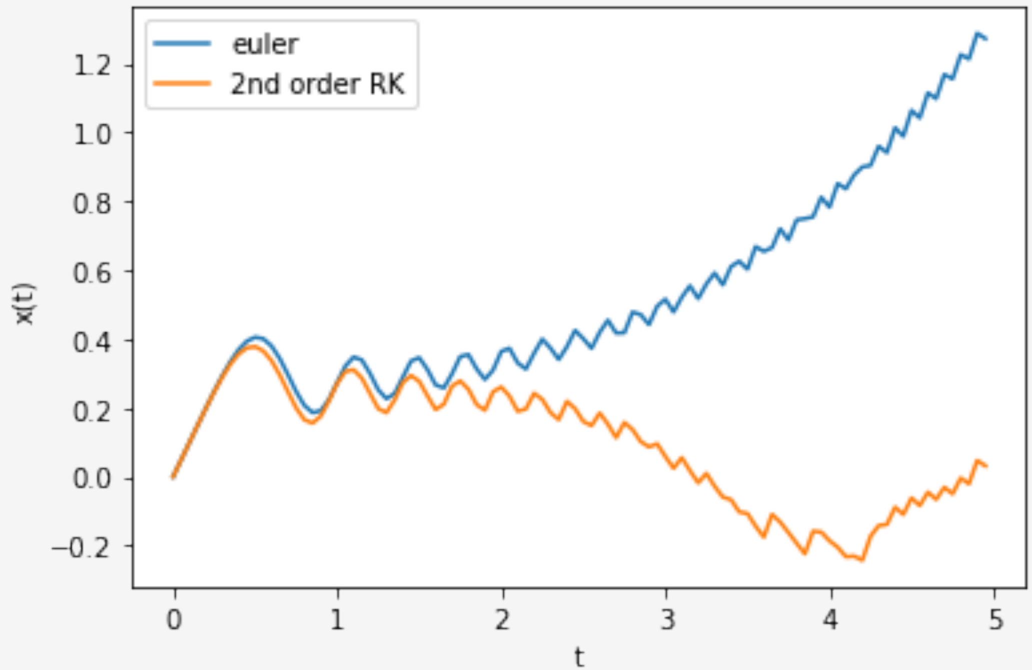


Similar

Second-order Runge-Kutta method, closer look

With small N, big differences

```
### Second-order Runge Kutta method vs Euler to solve dx/dt = e^(-x^2 + 3)*t^2 + cos(x^2*t^2) from t = 0 to t = 5 with x(0) = 0
### Smaller number of steps!
from math import cos,exp
from numpy import arange
from pylab import plot,xlabel,ylabel,show,legend

def f(x,t):
  return exp(-x**2+3)*t**2+cos(x**2*t**2)

a = 0.0 ## t=0 start
b = 5.0 ## max t
N = 100 ## number of steps
h = (b-a)/N ## size of single step
x_euler = 0.0 ## initial position
x_2rk = 0.0 ## initial position

tpoints = arange(a,b,h)
xpoints_2rk = []
xpoints_euler = []
for t in tpoints:
  xpoints_2rk.append(x_2rk)
  xpoints_euler.append(x_euler)
  k1 = h*f(x,t)
  k2 = h*f(x+0.5*k1,t+0.5*h)
  x_2rk += k2
  x_euler += h*f(x,t)

plot(tpoints,xpoints_euler,label="euler")
plot(tpoints,xpoints_2rk,label="2nd order RK")
xlabel("t")
ylabel("x(t)")
legend()
show()
```



Fourth-order Runge-Kutta method

Can use the same trick by Taylor expanding around additional points and taking combinations to cancel terms. The 4th-order Runge-Kutta method is a standard one to use (error of order h^5)

$$k_1 = hf(x,t)$$

$$k_2 = hf(x + \frac{k_1}{2}, t + \frac{h}{2})$$

$$k_3 = hf(x + \frac{k_2}{2}, t + \frac{h}{2})$$

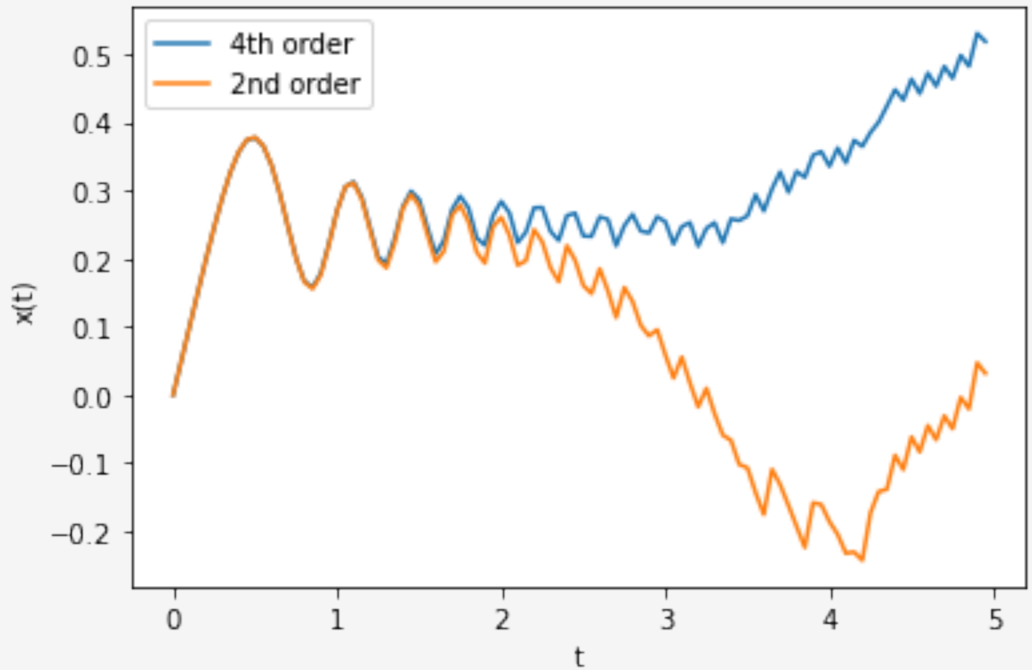
$$k_4 = hf(x + k_3, t + h)$$

$$x(t+h) = x(t) + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

Fourth-order Runge-Kutta method

With small N, big differences

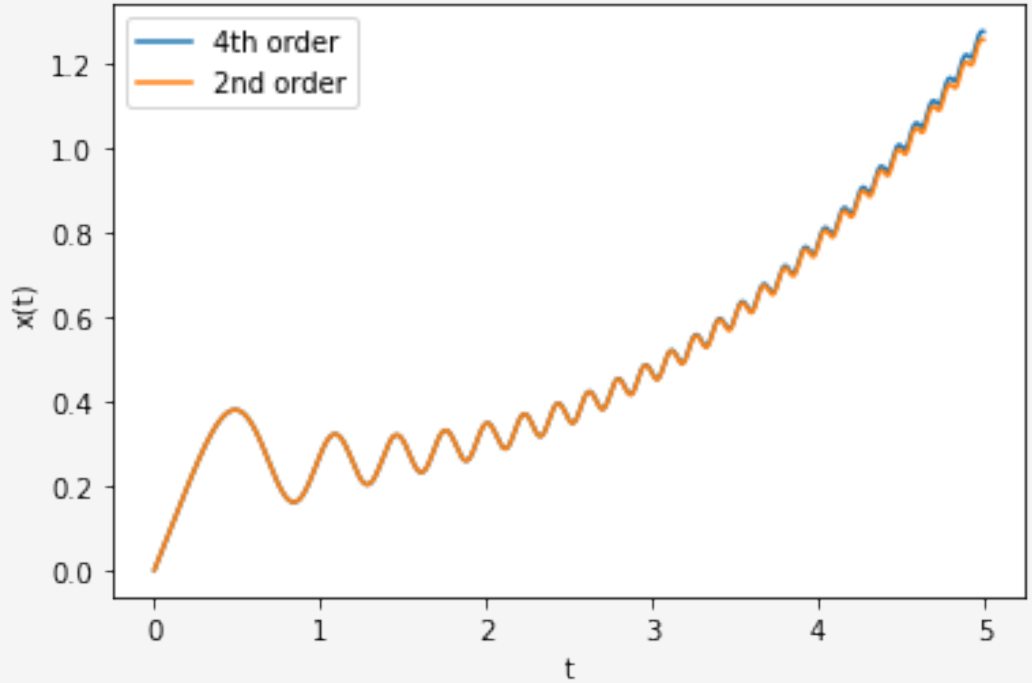
```
### Comparison of 2nd-order Runge Kutta method vs 4th order to solve dx/dt = e^(-x^2 + 3)*t^2 + cos(x^2*t^2) from t = 0 to t = 5 with x(0) = (

from math import cos,exp
from numpy import arange
from pylab import plot,xlabel,ylabel,show,legend

def f(x,t):
  return exp(-x**2+3)*t**2+cos(x**2*t**2)

a = 0.0 ## t=0 start
b = 5.0 ## max t
N = 100 ## number of steps
h = (b-a)/N ## size of single step
x_4 = 0.0 ## initial position
x_2 = 0.0 ## initial position

tpoints = arange(a,b,h)
xpoints_4 = []
xpoints_2 = []
for t in tpoints:
  xpoints_4.append(x_4)
  xpoints_2.append(x_2)
  k1 = h*f(x,t)
  k2 = h*f(x+0.5*k1,t+0.5*h)
  k3 = h*f(x+0.5*k2,t+0.5*h)
  k4 = h*f(x+k3,t+h)
  x_4 += (k1+2*k2+2*k3+k4)/6
  x_2 += k2

plot(tpoints,xpoints_4,label="4th order")
plot(tpoints,xpoints_2,label="2nd order")
xlabel("t")
ylabel("x(t)")
legend()
show()
```



Fourth-order Runge-Kutta method

With large N (small h), small differences

```
### Comparison of 2nd-order Runge Kutta method vs 4th order to solve dx/dt = e^(-x^2 + 3)*t^2 + cos(x^2*t^2) from t = 0 to t = 5 with x(0) = 0

from math import cos,exp
from numpy import arange
from pylab import plot,xlabel,ylabel,show,legend

def f(x,t):
  return exp(-x**2+3)*t**2+cos(x**2*t**2)

a = 0.0 ## t=0 start
b = 5.0 ## max t
N = 1000 ## number of steps
h = (b-a)/N ## size of single step
x_4 = 0.0 ## initial position
x_2 = 0.0 ## initial position

tpoints = arange(a,b,h)
xpoints_4 = []
xpoints_2 = []
for t in tpoints:
  xpoints_4.append(x_4)
  xpoints_2.append(x_2)
  k1 = h*f(x,t)
  k2 = h*f(x+0.5*k1,t+0.5*h)
  k3 = h*f(x+0.5*k2,t+0.5*h)
  k4 = h*f(x+k3,t+h)
  x_4 += (k1+2*k2+2*k3+k4)/6
  x_2 += k2

plot(tpoints,xpoints_4,label="4th order")
plot(tpoints,xpoints_2,label="2nd order")
xlabel("t")
ylabel("x(t)")
legend()
show()
```



Some additional thoughts

As textbook points out, we cannot take an infinite number of steps, but we can perform a change of variables such that $t = \infty$ corresponds to $u = 1$ and solve the equation for u and then transform back to t

And if we are interested in simultaneous first-order differential equations with one independent variable, we can still use the same formalism as before with $x \rightarrow \mathbf{r}$ and $k \rightarrow \mathbf{k}$

Finally, 2nd order differential equations can always be made into two 1st order differential equations, just as we can relate acceleration, velocity and time

Exercise 8.3, Lorenz equations

$$\frac{dx}{dt} = \sigma(y - x)$$

$$\frac{dy}{dt} = rx - y - xz$$

$$\frac{dz}{dt} = xy - bz$$

Studied by Edward Lorenz in the context of weather patterns - simplified model of atmospheric convection, a two-dimensional fluid layer warmed from below and cooled from above. x is the rate of convection, y the horizontal temperature variation and z the vertical temperature variation. r,σ and b are parameters of the system

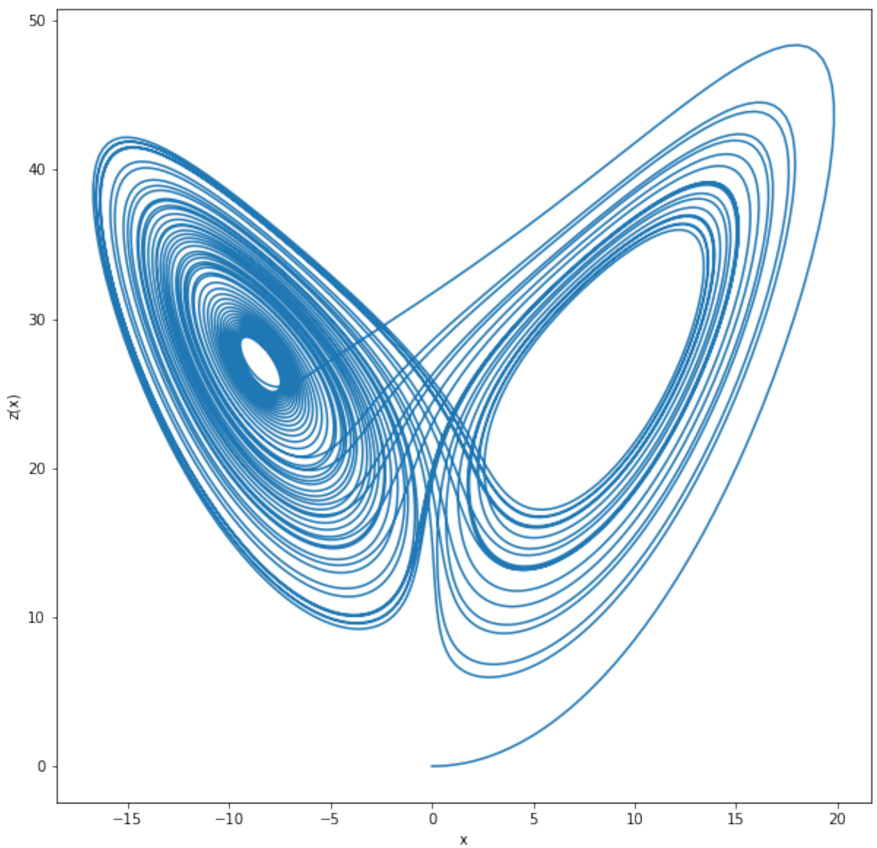
Exercise 8.3, Lorenz equations

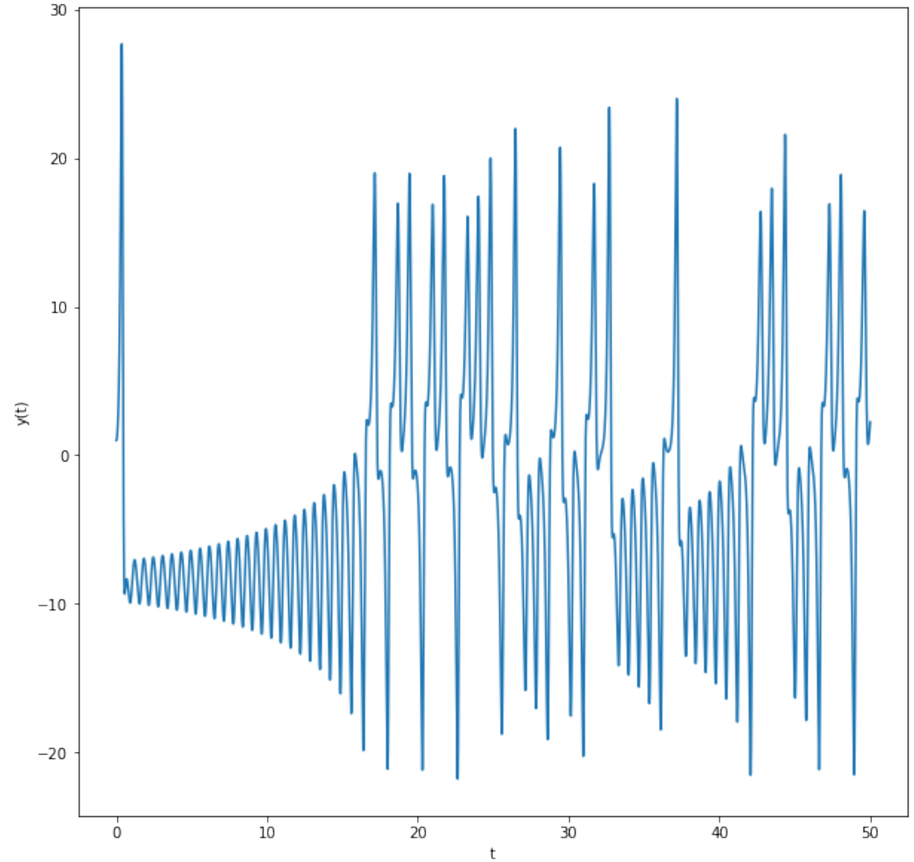

```
# Exercise 8.3, Lorenz Equations
from numpy import arange,array
from pylab import plot,xlabel,ylabel,show,figure,xlim,ylim


sigma = 10.0
r = 28.0
b = 8./3

def f(s,t):
  x = s[0]
  y = s[1]
  z = s[2]
  fx = sigma*(y-x)
  fy = r*x - y - x*z
  fz = x*y - b*z
  return array([fx,fy,fz],float)

N = 10000
tmin = 0.0
tmax = 50.0
h = (tmax - tmin)/N

s = array([0,1,0],float)
```

```
tpoints = arange(tmin,tmax,h)
xpoints = []
ypoints = []
zpoints = []

for t in tpoints:
  xpoints.append(s[0])
  ypoints.append(s[1])
  zpoints.append(s[2])

  k1 = h*f(s,t)
  k2 = h*f(s+0.5*k1,t+0.5*h)
  k3 = h*f(s+0.5*k2,t+0.5*h)
  k4 = h*f(s+k3,t+h)
  s += (k1+2*k2+2*k3+k4)/6


fig = figure(figsize=(10,10))
plot(xpoints,zpoints)
xlabel("x")
ylabel("z(x)")
show()

fig = figure(figsize=(10,10))
plot(tpoints,ypoints)
xlabel("t")
ylabel("y(t)")
show()
```

Solve with 4th order RK

Exercise 8.3, Lorenz equations





Very interesting behavior. On the left you see fixed points “attractors”. On the right you can see very bizarre, “chaotic” looking behavior

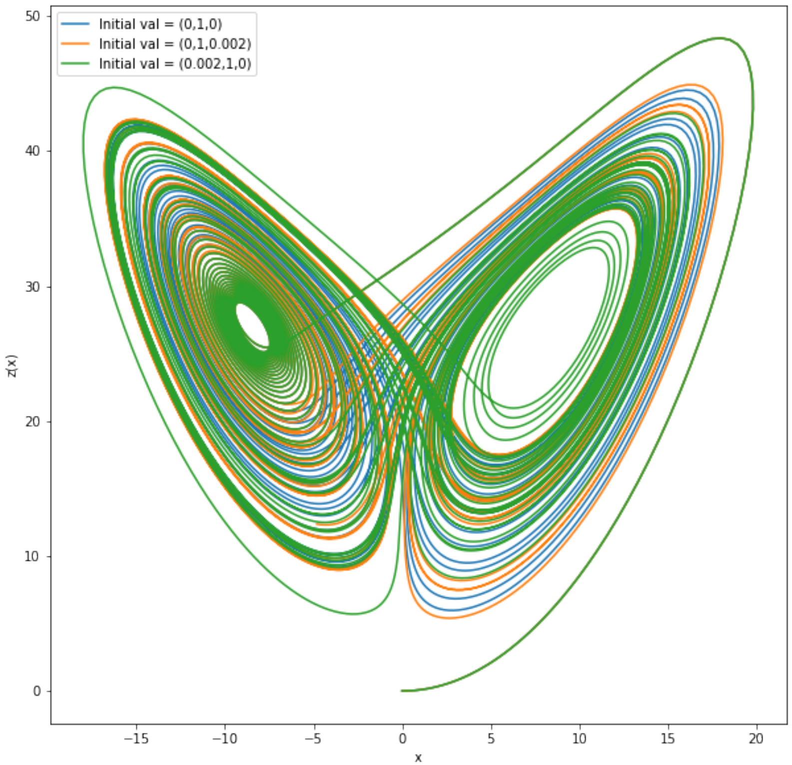
Lorenz equations, sensitivity to initial conditions

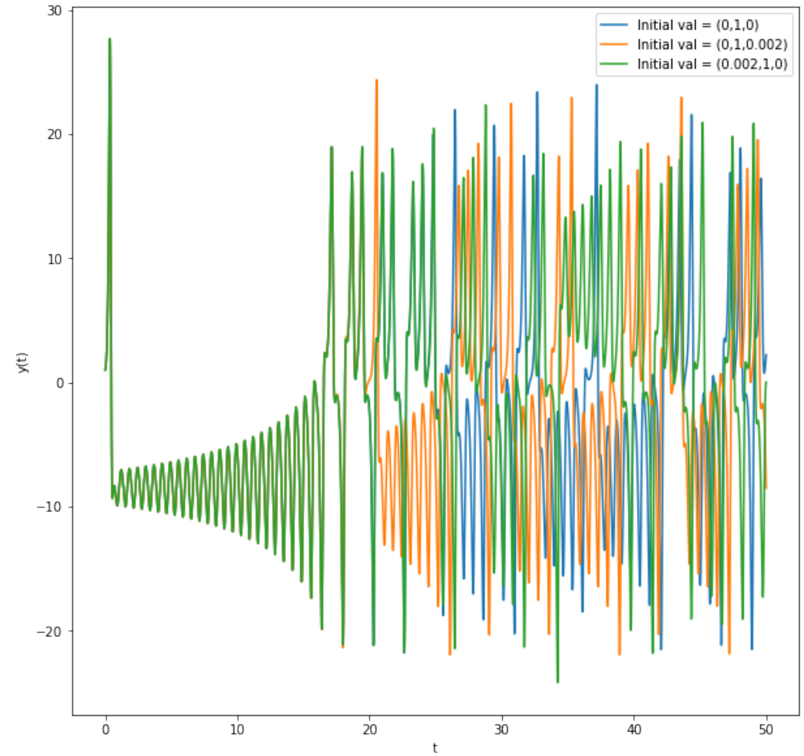
```
s = array([0,1,0],float)
s2 = array([0,1,0.002],float)
s3 = array([0.002,1,0],float)
```

```
diff2 = []
diff3 = []
for i in range(len(ypoints)):
  diff2.append(ypoints[i] - ypoints2[i])
  diff3.append(ypoints[i] - ypoints3[i])

fig = figure(figsize=(10,10))
plot(tpoints,diff2,label="Small dz initial")
plot(tpoints,diff3,label="Small dx initial")
xlabel("t")
ylabel("Delta(y(t)")
legend()
show()
```

Check sensitivity to initial conditions, give very small shifts in initial positions. What happens?

Lorenz equations, sensitivity to initial conditions

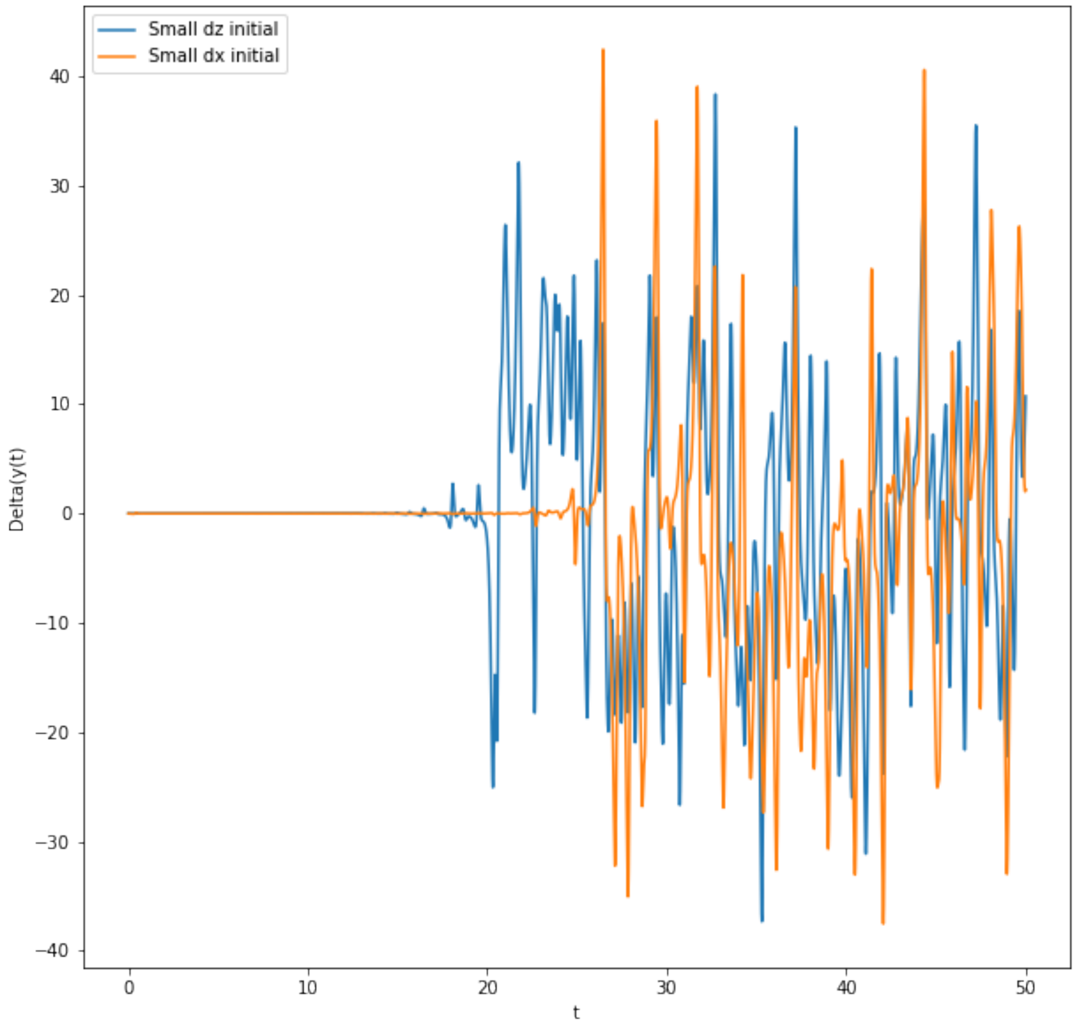




Very tiny shifts in initial conditions lead to wildly different behavior over time, even if the system is deterministic and the attractors are the same

Lorenz equations, sensitivity to initial conditions

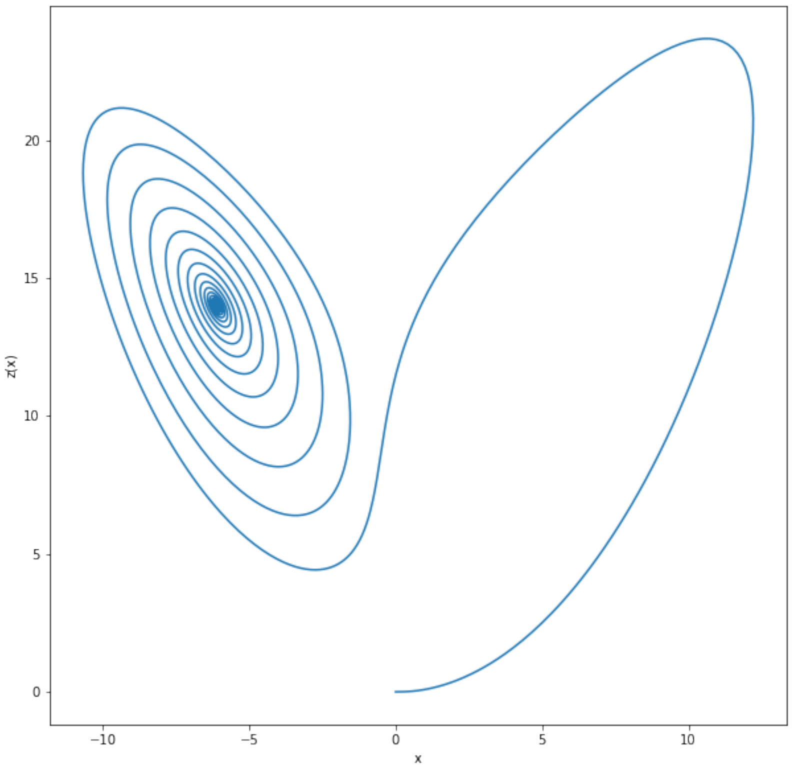
Very tiny shifts in initial conditions lead to wildly different behavior over time, even if the system is deterministic and the attractors are the same

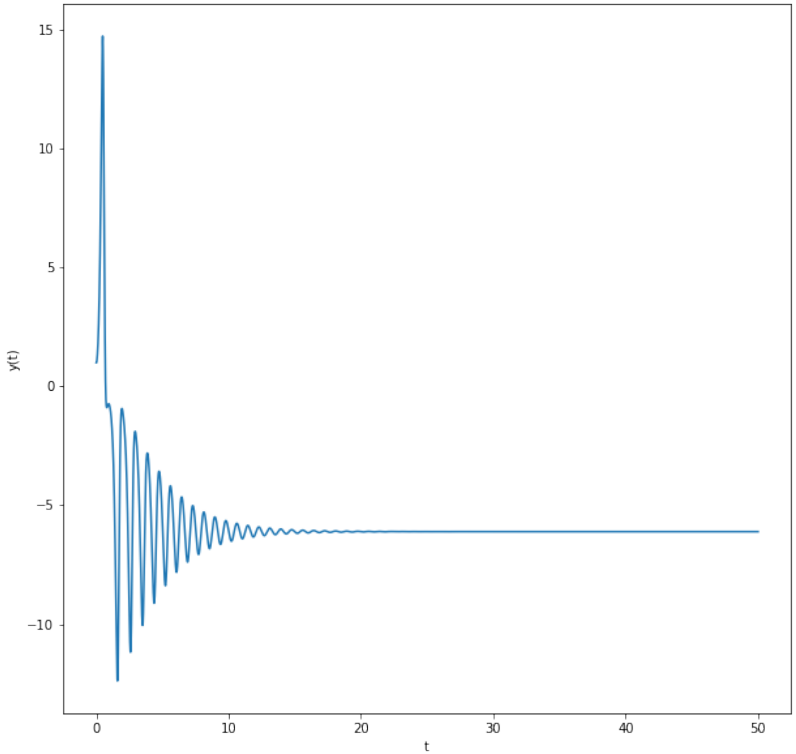


Lorenz equations, different rho

```
sigma = 10.0
r = 15.0
b = 8./3
```

Fun to play around with this





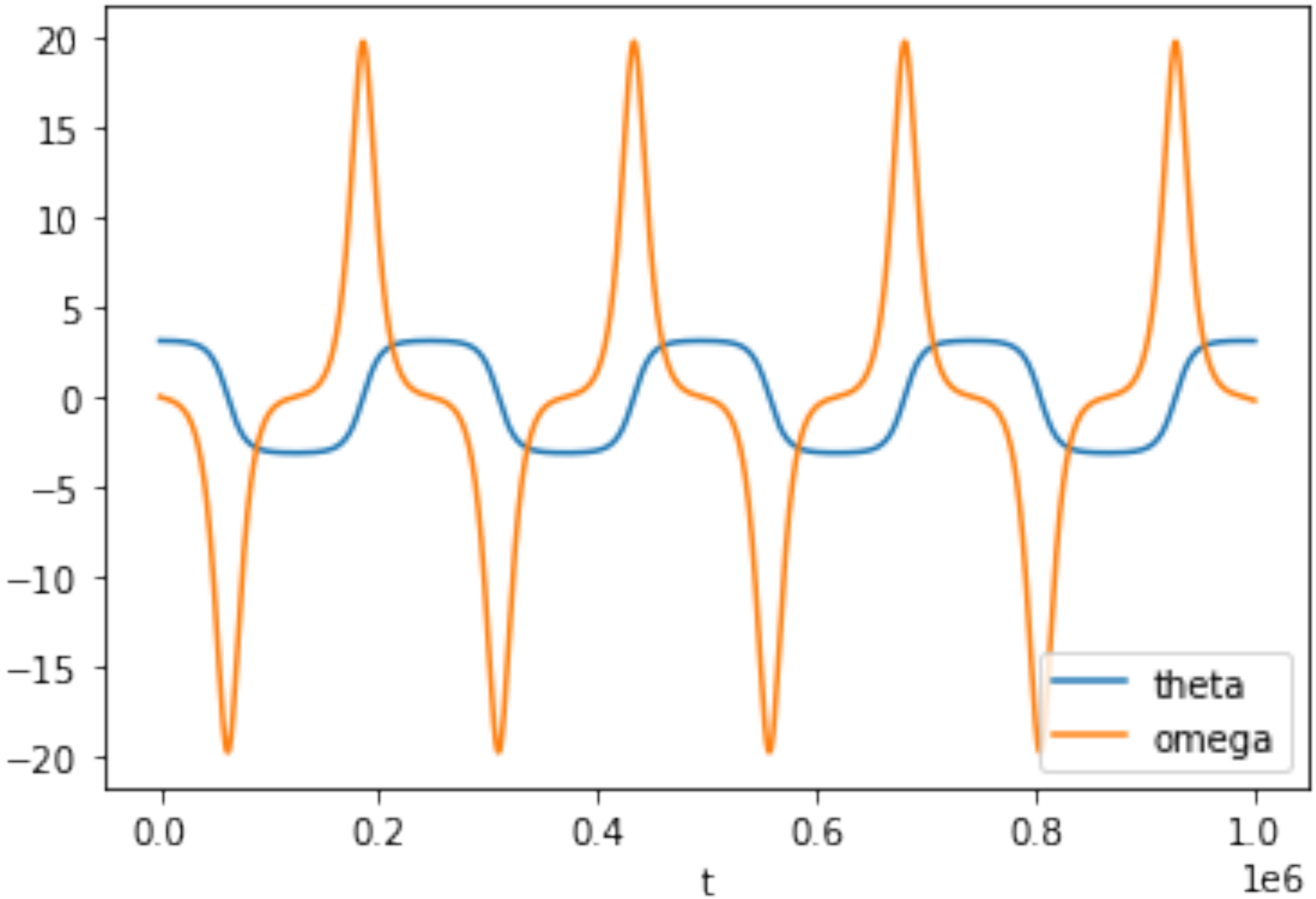
Exercise 8.4 Non-linear pendulum


```python
### Exercise 8.4 Non-linear pendulum

from math import sin,cos,pi
from numpy import array
from matplotlib.pyplot import plot,show,xlabel,ylabel

g = 9.81
l = 0.1
h = 1e-6

def f(r):
  theta = r[0]
  omega = r[1]
  ftheta = omega
  fomega = (-g/l)*sin(theta)
  return array([ftheta,fomega],float)
```

No simple analytic solution!

```python
# Initial values
r = array([pi*179/180,0.0],float)
Nstep = 1e6
# Main loop
thetas = []
omegas = []
step = 0
while step < Nstep:
  theta = r[0]
  thetas.append(theta)
  omegas.append(r[1])
  k1 = h*f(r)
  k2 = h*f(r+0.5*k1)
  k3 = h*f(r+0.5*k2)
  k4 = h*f(r+k3)
  r += (k1+2*k2+2*k3+k4)/6
  step = step+1

plot(thetas, label = "theta")
plot(omegas, label = "omega")
xlabel("t")
ylabel("theta")
legend()
show()
```

Exercise 8.4 Non-linear pendulum



Similar but definitely not identical behavior to linear pendulum

Adaptive Runge-Kutta method

The method we're using has an error that goes as h^5. Suppose we have a maximum target error, δ. We can check how well we are doing on a given step by making two steps of size h and then redoing it with a single step size of 2h. This second single step will by definition be less accurate, and we can compare:

$$x(t+2h) = x_1 + 2ch^5$$

c is unknown constant, 2 from two steps

$$x(t+2h) = x_2 + 32ch^5$$

c is same unknown constant, single factor of $(2h)^5$

Adaptive Runge-Kutta method

$$\epsilon = ch^5 = \frac{(x_1 - x_2)}{30}$$

Estimate of error on single step of size h

Our target error is δ, so target error per unit time is h'δ for some ideal step size h'. The target error on a single step is then

$$\epsilon^{'} = ch^{'5} = \frac{(x_1 - x_2)}{30}\left(\frac{h'}{h}\right)^5 = h'\delta$$

Adaptive Runge-Kutta method

$$\epsilon' = ch'^5 = \frac{(x_1 - x_2)}{30}\left(\frac{h'}{h}\right)^5 = h'\delta$$

$$h' = h\rho^4$$

$$\rho = \frac{30h\delta}{|x_1 - x_2|}$$

So at each step we use ρ to determine what our new step size should be (thus we place a limit on how big it can get in any given step). If solving simultaneous differential equations, need to think a little bit about what "error" means

Cometary Orbits (Exercise 8.10)

Comets orbit the sun in highly elliptical orbits, only force on them is due to gravity

$$m\frac{d^2\vec{r}}{dt^2} = -\left(\frac{GMm}{r^2}\right)\frac{\vec{r}}{r}$$

$$\frac{d^2x}{dt^2} = -GM\frac{x}{r^3}$$

$$\frac{d^2y}{dt^2} = -GM\frac{y}{r^3}$$

$$\frac{d^2z}{dt^2} = -GM\frac{z}{r^3}$$

Place orbital plane of comet perpendicular to the z axis so that z == 0 always and it’s a two-dimensional problem

Cometary Orbits (Exercise 8.10)

Comets orbit the sun in highly elliptical orbits, only force on them is due to gravity

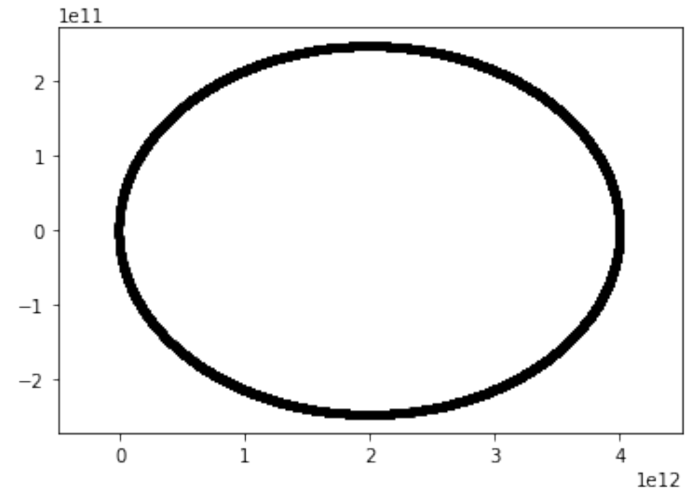
$$\frac{d^2x}{dt^2} = -GM\frac{x}{r^3}$$
$$\frac{d^2y}{dt^2} = -GM\frac{y}{r^3}$$

Convert two 2nd order equations into 4 1st order equations

$$\frac{dx}{dt} = u, \frac{dy}{dt} = v, \frac{du}{dt} = -GM\frac{x}{r^3}, \frac{dv}{dt} = -GM\frac{y}{r^3}$$

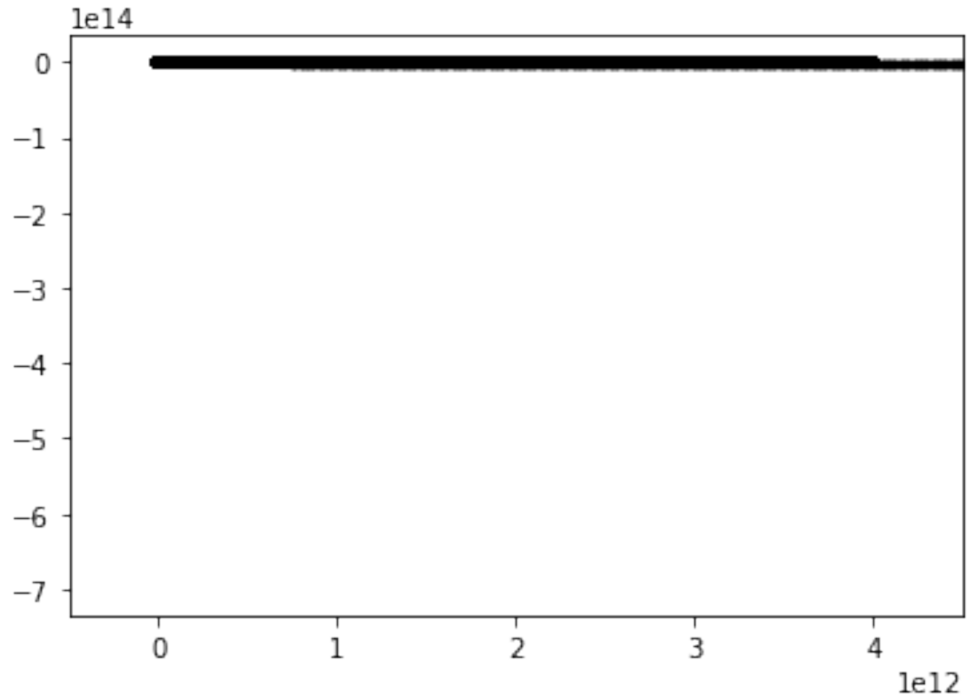
Cometary Orbits (Exercise 8.10)

```
### Exercise 8.10, using 4th order RK, fixed step size
from math import sqrt
from numpy import array
from pylab import plot,show


G = 6.674e-11 ## constant
M = 1.989e30 ## mass of sun in kg
h = 1.0e4 ### value of step size
tmax = 5.0e9 ## total time

x0 =  4.0e12 # starting position
y0 = 0.0

u0 = 0.0 ## starting velocity
v0 = 500.0

# Function
def f(r):
  x = r[0]
  y = r[1]
  u = r[2]
  v = r[3]
  rcubed = (x*x+y*y)**1.5
  fx = u
  fy = v
  fu = -G*M*x/rcubed
  fv = -G*M*y/rcubed
  return array([fx,fy,fu,fv],float)

# Setup the starting values
r = array([x0,y0,u0,v0],float)
xpoints = []
ypoints = []

t = 0.0
while t < tmax:
  k1 = h*f(r)
  k2 = h*f(r+0.5*k1)
  k3 = h*f(r+0.5*k2)
  k4 = h*f(r+k3)
  r += (k1+2*k2+2*k3+k4)/6
  xpoints.append(r[0])
  ypoints.append(r[1])
  t += h

# Make plot
plot(xpoints,ypoints,"k-")
plot(xpoints,ypoints,"k.")
xlim(-0.5e12,4.5e12)
show()
```



Non-adaptive code took 16 seconds to run (hover your mouse over the play button in Colab to see how long that snippet last took to run)

Cometary Orbits (Exercise 8.10)




```
h = 1.0e5 ### value of step size
```

Changing the step size by x10 leads to… well, nothing good

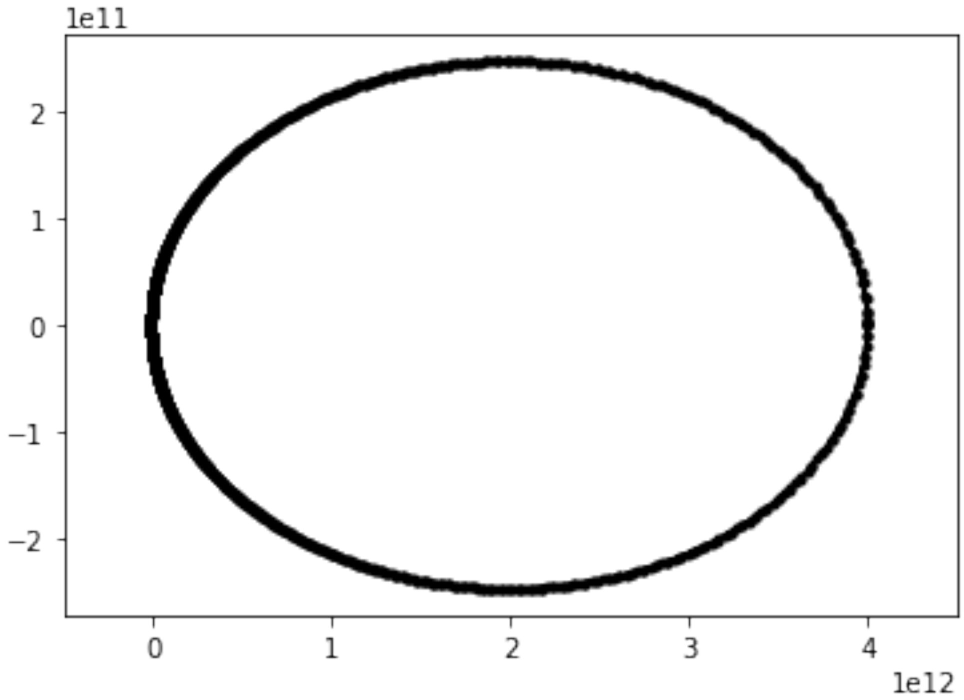
Cometary Orbits (Exercise 8.10), adaptive step size


```python
### Exercise 8.10, using 4th order RK, adaptive step size
from math import sqrt
from numpy import array
from pylab import plot,show


G = 6.674e-11 ## constant
M = 1.989e30 ## mass of sun in kg
h0 = 1.0e5 ### value of step size, initially very large
tmax = 5.0e9 ## total time
delta = 1e3/(365.25*24*3600) ## 1k/year (units of m/s)

x0 =  4.0e12 # starting position
y0 = 0.0

u0 = 0.0 ## starting velocity
v0 = 500.0

# Function
def f(r):
  x = r[0]
  y = r[1]
  u = r[2]
  v = r[3]
  rcubed = (x*x+y*y)**1.5
  fx = u
  fy = v
  fu = -G*M*x/rcubed
  fv = -G*M*y/rcubed
  return array([fx,fy,fu,fv],float)
```

```python
# Setup the starting values
r = array([x0,y0,u0,v0],float)
h = h0
xpoints = []
ypoints = []

t = 0.0
while t < tmax:
  # Do one large step

  k1 = 2*h*f(r)
  k2 = 2*h*f(r+0.5*k1)
  k3 = 2*h*f(r+0.5*k2)
  k4 = 2*h*f(r+k3)
  r1 = r + (k1+2*k2+2*k3+k4)/6

  # Do two small steps
  k1 = h*f(r)
  k2 = h*f(r+0.5*k1)
  k3 = h*f(r+0.5*k2)
  k4 = h*f(r+k3)
  r2 = r + (k1+2*k2+2*k3+k4)/6

  k1 = h*f(r2)
  k2 = h*f(r2+0.5*k1)
  k3 = h*f(r2+0.5*k2)
  k4 = h*f(r2+k3)
  r2 += (k1+2*k2+2*k3+k4)/6

  # Calculate rho
  dx = r1[0] - r2[0]
  dy = r1[1] - r2[1]
  rho = 30*h*delta/sqrt(dx*dx+dy*dy)

  # Calculate new t, h and r
  if rho >= 1.0: #can increase h, keep this point
    t += 2*h
    h *= min(rho**0.25,2.0) ### with cutoff
    r = r2
    xpoints.append(r[0])
    ypoints.append(r[1])
  else: ### nope, make h smaller, redo
    h *= rho ** 0.25

# Make plot
plot(xpoints,ypoints,"k-")
plot(xpoints,ypoints,"k.")
xlim(-0.5e12,4.5e12)
show()
```

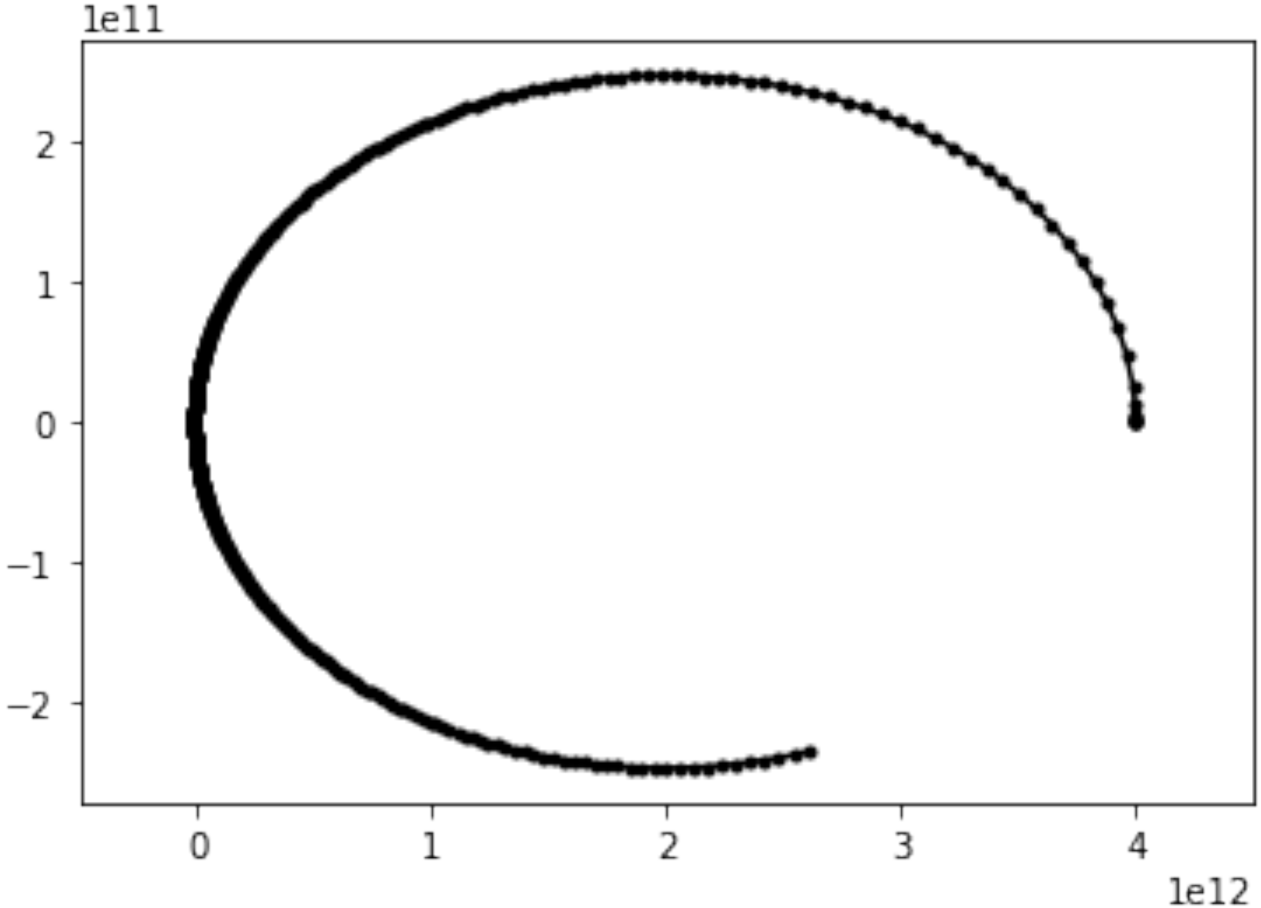
Cometary Orbits (Exercise 8.10), adaptive step size

Ran in 1.1 seconds! Order of magnitude faster



Cometary Orbits (Exercise 8.10), adaptive step size

Lower tmax to do < 1 orbit and see the adaptive step size in action



Repeating R-K from before

We had before (2nd order R-K):

$$x(t+h) = x(t) + hf(x(t+h/2), t+h/2)$$

$$x(t+h/2) = x(t) + \frac{h}{2} f(x,t)$$

Repeating:

$$x(t+2h) = x(t+h) + hf(x(t+\frac{3h}{2}), t+\frac{3h}{2})$$

$$x(t+\frac{3h}{2}) = x(t+h) + \frac{3h}{2} f(x, t+h)$$

Leapfrog method instead

Leapfrog method:

$$x(t+\frac{3h}{2}) = x(t+\frac{h}{2}) + hf(x(t+h), t+h)$$

And again:

$$x(t+\frac{4h}{2}) = x(t+\frac{2h}{2}) + hf(x(t+\frac{3h}{2}), t+\frac{3h}{2})$$

But note that we just calculated that in the previous step!

Two advantages: 1) It's time-reversal symmetric, and 2) It has error that is even in the step size h

Leapfrog method

Time-reversal symmetry: If we use the leapfrog method with step size (-h) we go backwards over our previous steps, so we can reverse time and get back to our initial state. NOT true of the RK method, necessarily. Time symmetry is critical in physics due to its connection to energy conservation. With the leapfrog method, the energy of the system is not a constant (of course, it should be, ideally), but it will come back to its initial value if the system is periodic. That is not the case with the RK method

Verlet method (not always possible!)

$$\ddot{x} = f(x,t), \dot{x} = v, \dot{v} = f$$

Leapfrog applied to both variables:

$$x(t+h) = x(t) + hv(t+\frac{h}{2})$$

$$v(t+\frac{3h}{2}) = v(t+\frac{h}{2}) + hf(x(t+h), t+h)$$

Thanks to structure of Newton's equations, we only need to know x(t+nh) and v(t+nh+1/2). And if we do need to know velocity at t+nh we can use Euler's method:

$$v(t+h) = v(t+\frac{h}{2}) + \frac{h}{2}f(x(t+h), t+h)$$

Modified Midpoint method

Going back to leapfrog method:

$$x(t+\frac{3h}{2}) = x(t+\frac{h}{2}) + hf(x(t+h), t+h)$$

Can consider these a set of estimates x(nt) and y(nt+1/2)

And then:

$$x(t+\frac{4h}{2}) = x(t+\frac{2h}{2}) + hf(x(t+\frac{3h}{2}), t+\frac{3h}{2})$$

$$x_0 = x(t_0)$$

$$y_1 = x_0 + \frac{h}{2} f(x_0, t)$$

$$x_1 = x_0 + hf(y_1, t+\frac{h}{2})$$

$$y_2 = y_1 + hf(x_1, t+h)$$

$$x_2 = x_1 + hf(y_2, t+\frac{3h}{2})$$

$$y_3 = y_2 + hf(x_2, t+2h)$$

Or more generally:

$$y_{m+1} = y_m + hf(x_m, t+mh)$$

$$x_{m+1} = x_m + hf(y_{m+1}, t+(m+\frac{1}{2})h)$$

Modified Midpoint method

Imagine we take n total steps to arrive at final point at time t+H = x_n

But we can also use y_n and Euler's method with step size h/2 to calculate the final point:

$$x(t+H) = y_n + \frac{h}{2} f(x_n, t+H)$$

So let's average the two (not clear one is better):

$$x(t+H) = \frac{1}{2}\left(x_n + y_n + \frac{h}{2} f(x_n, t+H)\right)$$

It turns out that this is very useful (only has errors that contain even powers of h)

Let's imagine we're only interested in the solution at some later final time t_0+H

We start with a crude estimate from the modified midpoint method, which is obviously not a very good estimate unless H is very small

$$R_{11} = x(t+H) = \frac{1}{2}\left(x_n + y_n + \frac{h}{2}f(x_n, t+H)\right)$$

Now we redo the method again with the midpoint method using a step size h_2 = H/2 giving us an estimate R_{21} for x(t+H)

Bulirsch-Stoer method

Since the errors are purely in even powers of h, we have no h^3 term (nice!) and we get:

$$x(t+H) = R_{21} + c_1 h_2^2 + \mathcal{O}(h_2^4)$$

$$x(t+H) = R_{21} + c_1 h_2^2 + \mathcal{O}(h_2^4) = R_{11} + c_1 h_1^2 + \mathcal{O}(h_2^4)$$

$$x(t+H) = R_{21} + c_1 h_2^2 + \mathcal{O}(h_2^4) = R_{11} + 4c_1 h_2^2 + \mathcal{O}(h_2^4)$$

$$c_1 h_2^2 = \frac{1}{3}(R_{21} - R_{11})$$

Substitute back in…

$$x(t+H) = R_{22} = R_{21} + \frac{1}{3}(R_{21} - R_{11}) + \mathcal{O}(h_2^4)$$

Now error goes as h^4! Can keep iterating this process

Bulirsch-Stoer method

View this a series expansion of our estimate in powers of h, and we stop when our error is small enough. This idea of Richardson extrapolation is very similar to Romberg integration. If we want estimates at more than time t_0+H we can repeat the method multiple times. Because it is an expansion the solution has to be relatively smooth for this to work

Boundary value problems

Consider a differential equation like this one (we have seen and will continue to see many examples of it), with boundaries x_0 and x_1

$$\frac{d^2u}{dx^2} = f(x, u, u')$$

$$u' = \frac{du}{dx}$$

Dirichlet: Specify the values u(x) on the two boundaries:

Neumann: Specify u'(x) on the two boundaries

Periodic: Specify $u(x_0) = u(x_1)$, $u'(x_0) = u'(x_1)$

Or we can have a combination of the above

Familiar boundary value problem

$$-\frac{\hbar^2}{2m}\frac{d^2\psi}{dx^2} + V(x)\psi(x) = E\psi(x)$$

If we have an infinite potential well, V(x) = 0 for x < L and infinite otherwise, thus we get Dirichlet boundary conditions $\psi(0) = \psi(\mathrm{L}) = 0$

Shooting method for solving boundary value problems

Just do a trial-and-error approach by formulating the boundary value problem as finding the root of an equation, which we know how to do (binary search method, secant method). In other words, we find a solution to our equation that **does not** match our boundary conditions and keep modifying our solution to find one that does

Exercise 8.14 Quantum Oscillators

$$-\frac{\hbar^2}{2m}\frac{d^2\psi}{dx^2} + V(x)\psi(x) = E\psi(x)$$

Hopefully something you've seen before :) We can put this in a form that we can use with:

$$\frac{d\psi}{dx} = \phi(x)$$

$$\frac{d\phi}{dx} = \frac{2m}{\hbar^2}\left[V(x) - E\right]\psi(x)$$

What are the energies of this system if V(x) = V0(x^2/a^2)

Exercise 8.14 Quantum Oscillators

```
from math import sqrt
from numpy import array,arange,linspace
from pylab import plot,show,xlim,ylim

# Constants
m = 9.1094e-31 # mass of electron in kg
hbar = 1.0546e-34 # hbar
e = 1.6022e-19 # electron charge in C

a = 1e-11 # width parameter of the well
V0 = 50*e # Multiplier
w = 10*a ### We need a boundary for the wave function, it is at infinity but we can try 10*a
N = 1000 ## iterations in x for solver
h = 2*w/N

# Potential function
def Vharmonic(x):
  return V0*(x/a)**2

# Potential function
def VAharmonic(x):
  return V0*(x/a)**4

def f(r,x,E,V):
  psi = r[0]
  phi = r[1]
  fpsi = phi
  fphi = (2*m/hbar**2)*(V(x)-E)*psi
  return array([fpsi,fphi],float)
```

Can'y go out to infinity, just pick large value. You can check the effect of this!

```
# Function to calculate the wave function for any potential function
def solve(E,V):
  psi = 0.0
  phi = 1.0
  r = array([psi,phi],float)
  psipoints = [ psi ]

  for x in arange(-w,w,h):
    k1 = h*f(r,x,E,V)
    k2 = h*f(r+0.5*k1,x+0.5*h,E,V)
    k3 = h*f(r+0.5*k2,x+0.5*h,E,V)
    k4 = h*f(r+k3,x+h,E,V)
    r += (k1+2*k2+2*k3+k4)/6
    psipoints.append(r[0])

  return array(psipoints,float)

# Function to return the value of psi at the boundary, check only one side, it's symmetric
def boundary(E,V):
  psi = solve(E,V)
  return psi[N]
```

Exercise 8.14 Quantum Oscillators

```
# Function to find the energy using the secant method
def secant(E1,E2,V):
  psi2 = boundary(E1,V)
  target = e/1000
  while abs(E1-E2) > target:
    psi1,psi2 = psi2,boundary(E2,V)
    E1,E2 = E2,E2-psi2*(E2-E1)/(psi2-psi1)
  return E2

# Function to plot normalized wave function at a certain energy
def waveplot(E,V):
  # Calculate the unnormalized wavefunction
  psi = solve(E,V)
  halfpsi = psi[0:N//2+1] ### use the fact that it's symmetric

  # Normalize it, roughly
  integral = 2*h*(sum(halfpsi**2) - 0.5*halfpsi[0]**2 - 0.5*halfpsi[N//2]**2)
  psi /= sqrt(integral)
  x = linspace(-w,w,N+1)
  plot(x,psi)
```

Secant method needs some initial values

Just a rough normalization

Exercise 8.14 Quantum Oscillators

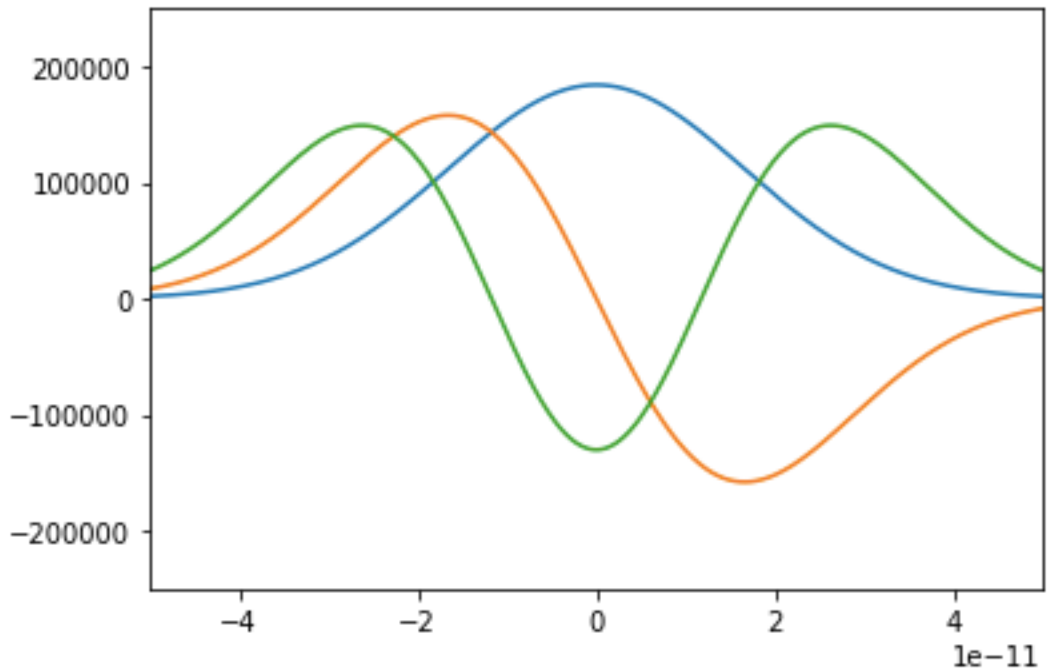

```
print("Aharmonic:")
V = VAharmonic
E = secant(0*e,10*e,V)
print(E/e)
waveplot(E,V)

E = secant(500*e,510*e,V)
print(E/e)
waveplot(E,V)

E = secant(1000*e,1010*e,V)
print(E/e)
waveplot(E,V)

xlim(-5*a,5*a)
ylim(-250000,250000)
show()
```

```
print("harmonic:")
V = Vharmonic
E = secant(0*e,10*e,V)
print(E/e)
waveplot(E,V)

E = secant(200*e,210*e,V)
print(E/e)
waveplot(E,V)

E = secant(500*e,510*e,V)
print(E/e)
waveplot(E,V)

xlim(-5*a,5*a)
ylim(-250000,250000)
show()
```

Need to provide some starting values for the secant method. And we have to shift the starting values to get the different energy levels! Could try to automate this

Exercise 8.14 Quantum Oscillators

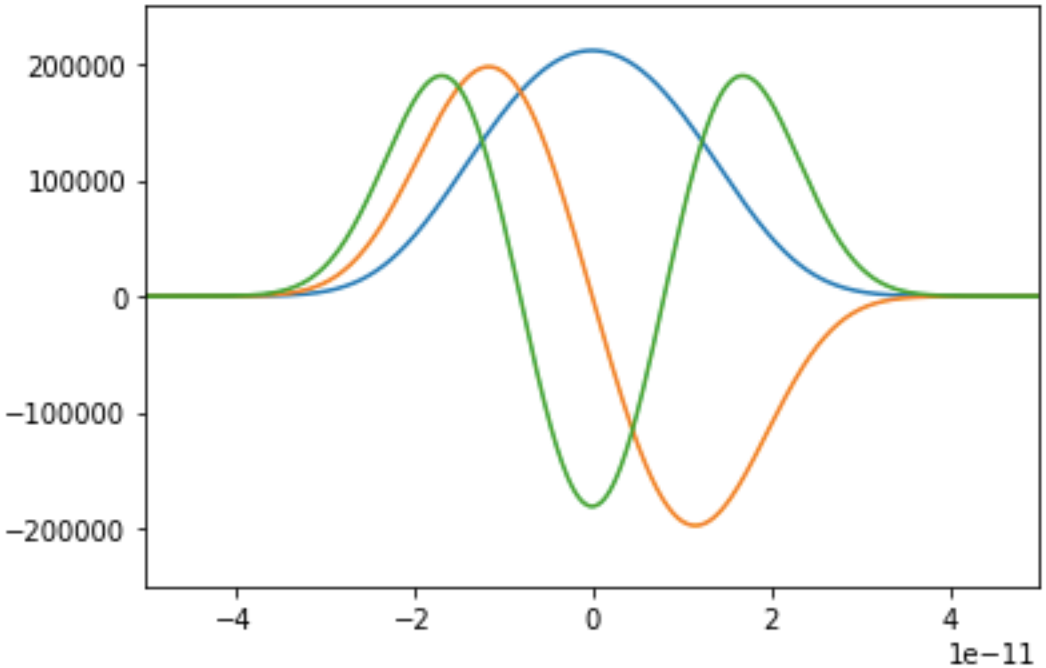

```
harmonic:
138.02397191032284
414.07191730732234
690.1198620526808
```



Energy values are roughly equally spaced (276 eV), can check this

Exercise 8.14 Quantum Oscillators


```
Aharmonic:
205.3069034694279
735.6912470013569
1443.569421326702
```



What if $V(x) = V0(x^4/a^4)$? Energies not equally spaced!!!

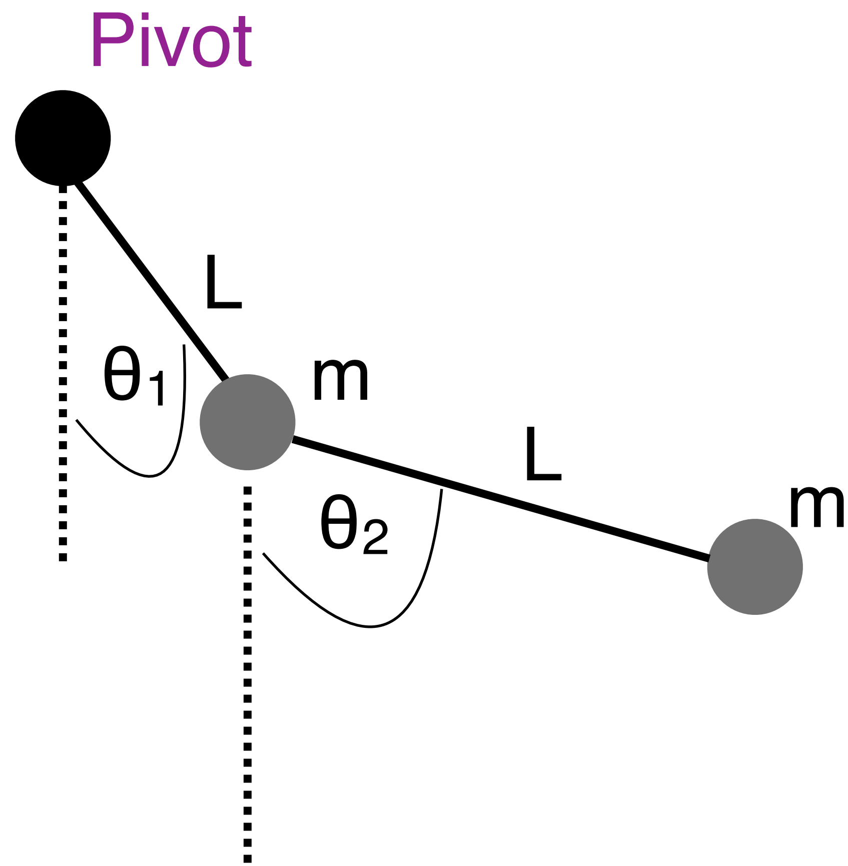
A word on visualization

The book uses the “visual” package of python. This is an out-of-date reference. The up-to-date package is called vpython. To make matters worse, the interfaces and uses have changed in the newer version. If you can use the old version, great. Otherwise, use the new version. Otherwise, just use matplotlib instead, though it doesn’t do the full 3d graphics

One other challenge with vpython - saving animations, even not inside colab/notebooks, is not so easy to do :(

Let’s look at Exercise 8.15, first using vpython and then using matplotlib

Double pendulum



$$h_1 = -L\cos\theta_1$$

$$h_2 = -L(\cos\theta_1 + \cos\theta_2)$$

$$V = mgh_1 + mgh_2 = -mgL(2\cos\theta_1 + \cos\theta_2)$$

$$v_1 = L\dot{\theta}_1$$

Double pendulum

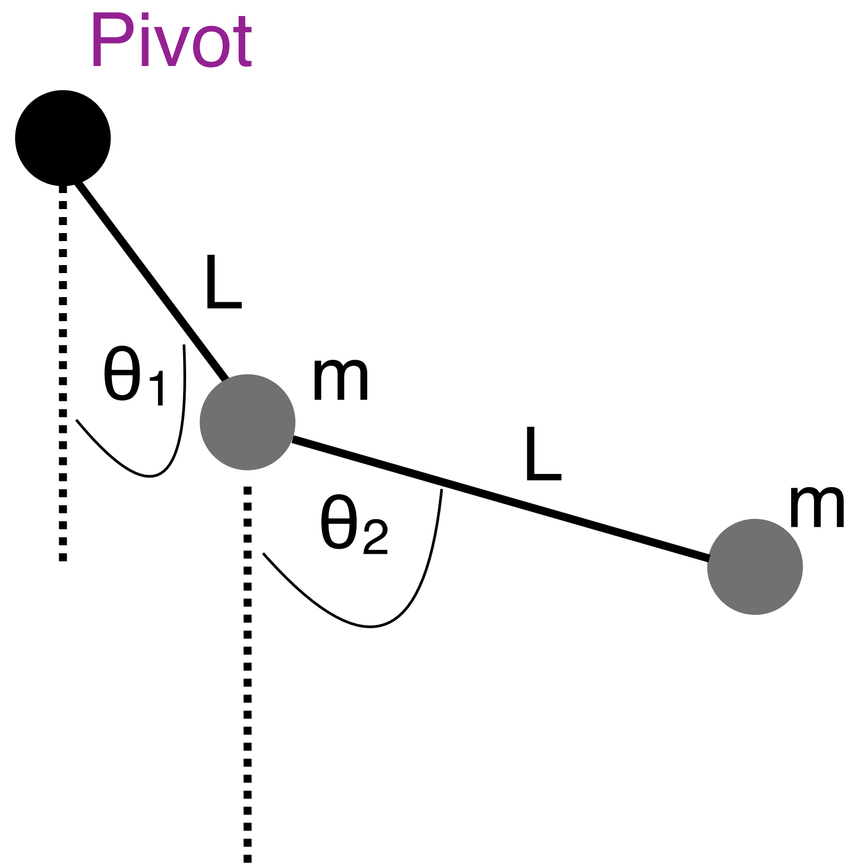
$$x_2 = L\sin\theta_1 + L\sin\theta_2$$

$$y_2 = -L\cos\theta_1 - L\cos\theta_2$$

$$v_{2,x} = L\dot{\theta}_1\cos\theta_1 + L\dot{\theta}_2\cos\theta_2$$

$$v_{2,y} = L\dot{\theta}_1\sin\theta_1 + L\dot{\theta}_2\sin\theta_2$$

$$v_2 = \sqrt{v_{2,x}^2 + v_{2,y}^2}$$

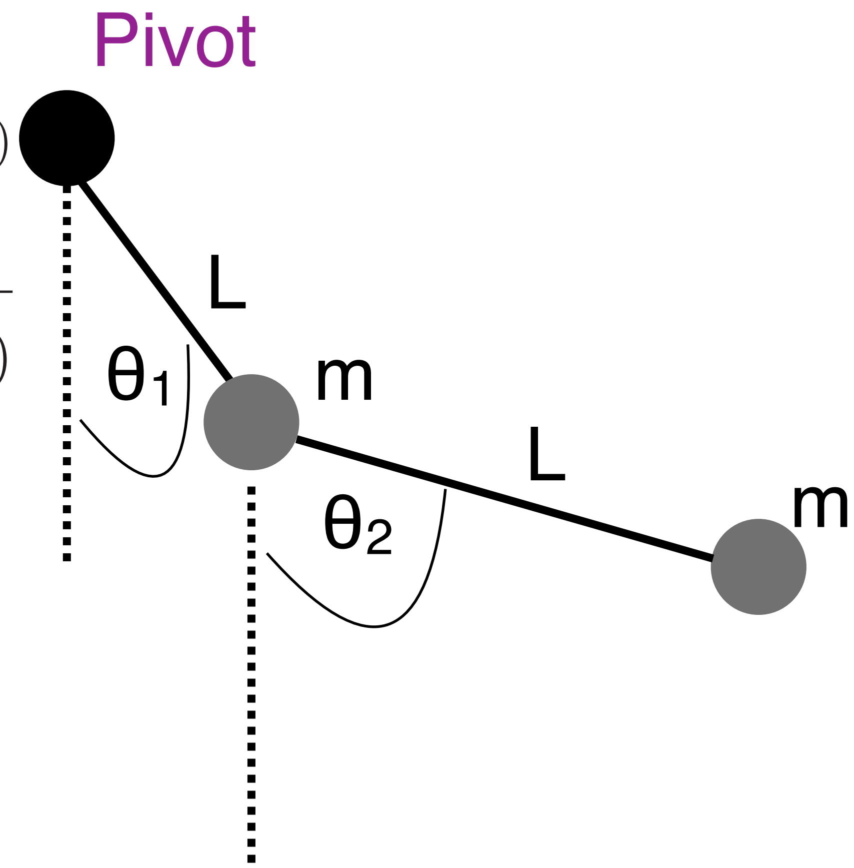


$$v_2 = \sqrt{L^2\dot{\theta}_1^{\,2}\cos^2\theta_1 + L^2\dot{\theta}_2^{\,2}\cos^2\theta_2 + 2L^2\dot{\theta}_1\cos\theta_1\dot{\theta}_2\cos\theta_2 + L^2\dot{\theta}_1^{\,2}\sin^2\theta_1 + L^2\dot{\theta}_2^{\,2}\sin^2\theta_2 + 2L^2\dot{\theta}_1\sin\theta_1\dot{\theta}_2\sin\theta}$$

$$v_2 = L\sqrt{\dot{\theta}_1^{\,2}(\cos^2\theta_1 + \sin^2\theta_1) + \dot{\theta}_2^{\,2}(\cos^2\theta_2 + \sin^2\theta_2) + 2\dot{\theta}_1\dot{\theta}_2(\cos\theta_1\cos\theta_2 + \sin\theta_1\sin\theta_2)}$$

$$v_2 = L\sqrt{\dot{\theta}_1^{\,2} + \dot{\theta}_2^{\,2} + 2\dot{\theta}_1\dot{\theta}_2(\cos(\theta_1 - \theta_2))}$$

Double pendulum



$$V = mgh_1 + mgh_2 = -mgL(2\cos\theta_1 + \cos\theta_2)$$

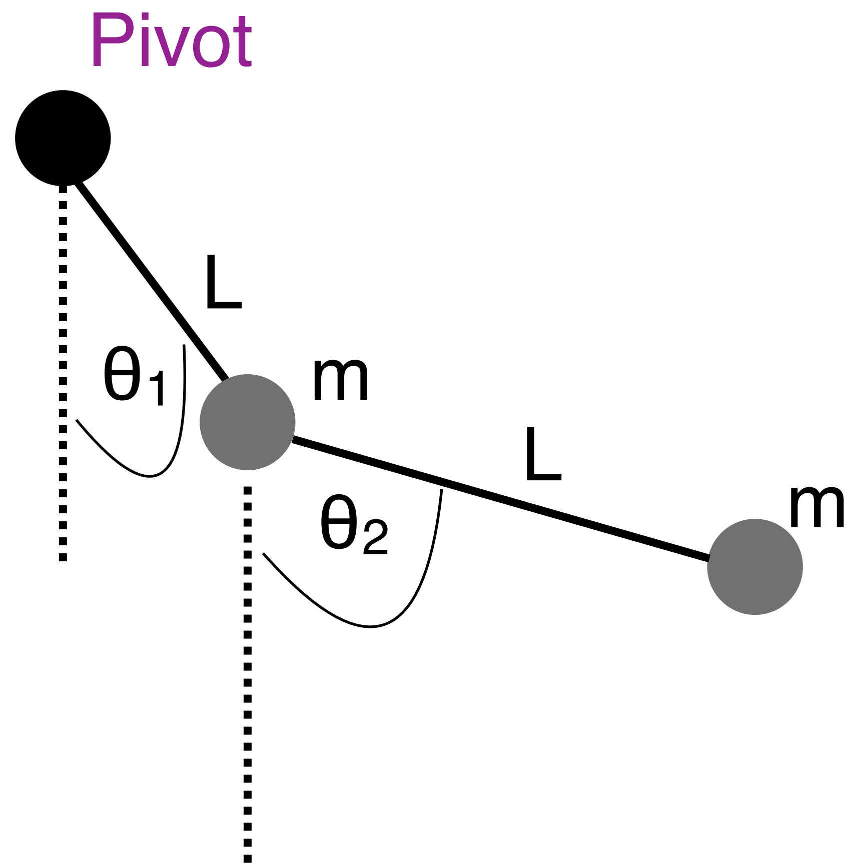
$$v_1 = L\dot{\theta}_1$$

$$v_2 = L\sqrt{\dot{\theta}_1{}^2 + \dot{\theta}_2{}^2 + 2\dot{\theta}_1\dot{\theta}_2(\cos(\theta_1 - \theta_2))}$$

$$v_1^2 = L^2\dot{\theta}_1{}^2$$

$$v_2^2 = L^2\left(\dot{\theta}_1{}^2 + \dot{\theta}_2{}^2 + 2\dot{\theta}_1\dot{\theta}_2(\cos(\theta_1 - \theta_2))\right)$$

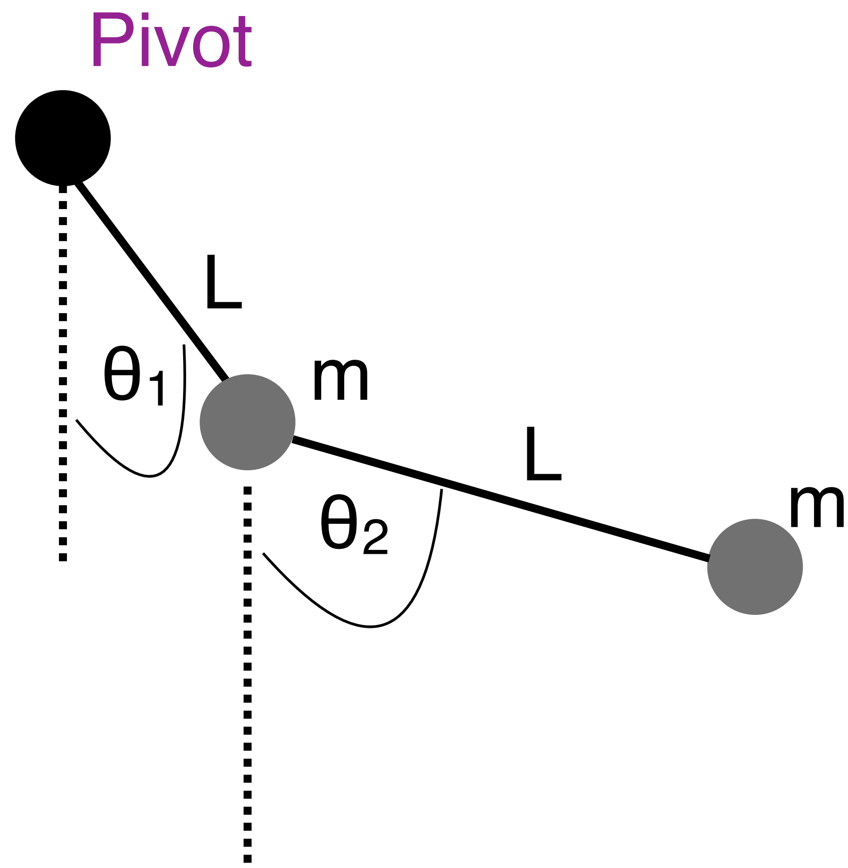
Double pendulum



$$T = \frac{1}{2}m(v_1^2 + v_2^2) = \frac{mL^2}{2}\left(2\dot{\theta_1}^2 + \dot{\theta_2}^2 + 2\dot{\theta}_1\dot{\theta}_2(\cos(\theta_1 - \theta_2))\right)$$

$$V = mgh_1 + mgh_2 = -mgL(2\cos\theta_1 + \cos\theta_2)$$

Double pendulum



$$\frac{\partial \mathcal{L}}{\partial \dot{\theta}_1} = 2mL^2\dot{\theta}_1 + mL^2\dot{\theta}_2\cos(\theta_1 - \theta_2)$$

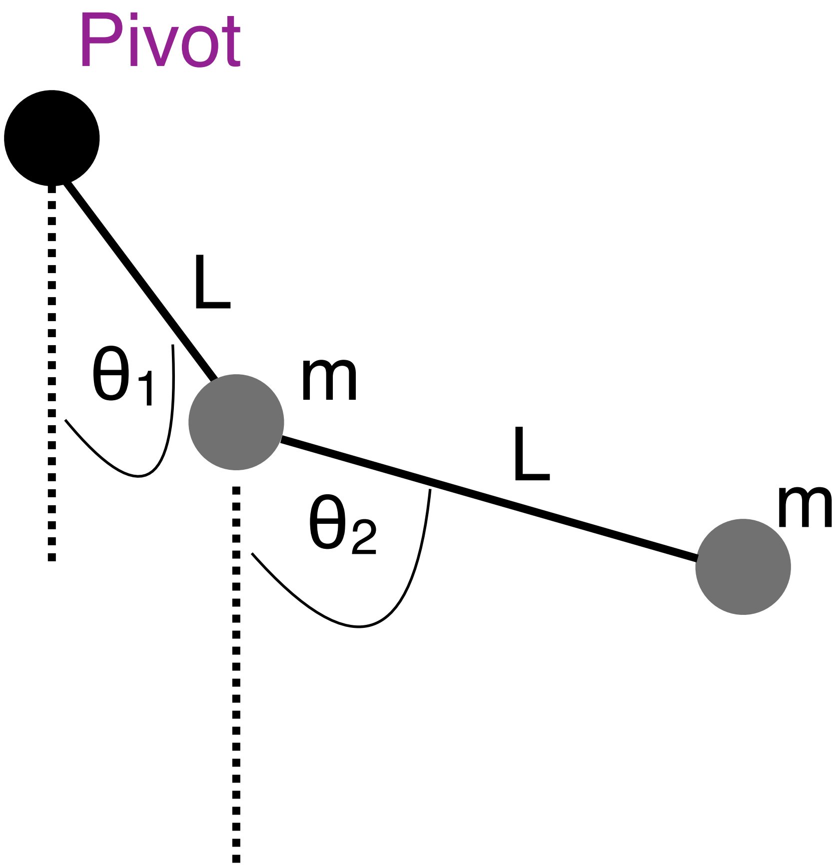
$$\frac{\partial \mathcal{L}}{\partial \dot{\theta}_2} = mL^2\dot{\theta}_2 + mL^2\dot{\theta}_1\cos(\theta_1 - \theta_2)$$

$$\frac{\partial \mathcal{L}}{\partial \theta_1} = -mL^2\dot{\theta}_1\dot{\theta}_2\sin(\theta_1 - \theta_2) - 2mgL\sin\theta_1$$

$$\frac{\partial \mathcal{L}}{\partial \theta_2} = mL^2\dot{\theta}_1\dot{\theta}_2\sin(\theta_1 - \theta_2) - mgL\sin\theta_2$$

$$\mathcal{L} = T - U = \frac{mL^2}{2}\left(2\dot{\theta}_1{}^2 + \dot{\theta}_2{}^2 + 2\dot{\theta}_1\dot{\theta}_2(\cos(\theta_1 - \theta_2))\right) + mgL\left(2\cos\theta_1 + \cos\theta_2\right)$$

Double pendulum



$$\frac{\partial \mathcal{L}}{\partial \dot\theta_1} = 2mL^2\dot\theta_1 + mL^2\dot\theta_2\cos(\theta_1-\theta_2)$$

$$\frac{\partial \mathcal{L}}{\partial \dot\theta_2} = mL^2\dot\theta_2 + mL^2\dot\theta_1\cos(\theta_1-\theta_2)$$

$$\frac{\partial \mathcal{L}}{\partial \theta_1} = -mL^2\dot\theta_1\dot\theta_2\sin(\theta_1-\theta_2) - 2mgL\sin\theta_1$$

$$\frac{\partial \mathcal{L}}{\partial \theta_2} = mL^2\dot\theta_1\dot\theta_2\sin(\theta_1-\theta_2) - mgL\sin\theta_2$$

Euler-Lagrange!

$$2mL^2\ddot\theta_1 + mL^2\ddot\theta_2\cos(\theta_1-\theta_2) + mL^2\dot\theta_2\frac{d}{dt}\cos(\theta_1-\theta_2) = -mL^2\dot\theta_1\dot\theta_2\sin(\theta_1-\theta_2) - 2mgL\sin\theta_1$$

$$mL^2\ddot\theta_2 + mL^2\ddot\theta_1\cos(\theta_1-\theta_2) + mL^2\dot\theta_1\frac{d}{dt}\cos(\theta_1-\theta_2) = mL^2\dot\theta_1\dot\theta_2\sin(\theta_1-\theta_2) - mgL\sin\theta_2$$

Double pendulum

$$2mL^2\ddot{\theta}_1 + mL^2\ddot{\theta}_2\cos(\theta_1-\theta_2) + mL^2\dot{\theta}_2\frac{d}{dt}\cos(\theta_1-\theta_2) = -mL^2\dot{\theta}_1\dot{\theta}_2\sin(\theta_1-\theta_2) - 2mgL\sin\theta_1$$

$$mL^2\ddot{\theta}_2 + mL^2\ddot{\theta}_1\cos(\theta_1-\theta_2) + mL^2\dot{\theta}_1\frac{d}{dt}\cos(\theta_1-\theta_2) = mL^2\dot{\theta}_1\dot{\theta}_2\sin(\theta_1-\theta_2) - mgL\sin\theta_2$$

$$2L\ddot{\theta}_1 + L\ddot{\theta}_2\cos(\theta_1-\theta_2) + L\dot{\theta}_2\frac{d}{dt}\cos(\theta_1-\theta_2) = -L\dot{\theta}_1\dot{\theta}_2\sin(\theta_1-\theta_2) - 2g\sin\theta_1$$

$$L\ddot{\theta}_2 + L\ddot{\theta}_1\cos(\theta_1-\theta_2) + L\dot{\theta}_1\frac{d}{dt}\cos(\theta_1-\theta_2) = L\dot{\theta}_1\dot{\theta}_2\sin(\theta_1-\theta_2) - g\sin\theta_2$$

$$2L\ddot{\theta}_1 + L\ddot{\theta}_2\cos(\theta_1-\theta_2) + L\dot{\theta}_2\sin(\theta_1-\theta_2)(\dot{\theta}_2-\dot{\theta}_1) = -L\dot{\theta}_1\dot{\theta}_2\sin(\theta_1-\theta_2) - 2g\sin\theta_1$$

$$L\ddot{\theta}_2 + L\ddot{\theta}_1\cos(\theta_1-\theta_2) + L\dot{\theta}_1\sin(\theta_1-\theta_2)(\dot{\theta}_2-\dot{\theta}_1) = L\dot{\theta}_1\dot{\theta}_2\sin(\theta_1-\theta_2) - g\sin\theta_2$$

$$2L\ddot{\theta}_1 + L\ddot{\theta}_2\cos(\theta_1-\theta_2) + L\dot{\theta}_2^{\,2}\sin(\theta_1-\theta_2) = -2g\sin\theta_1$$

$$L\ddot{\theta}_2 + L\ddot{\theta}_1\cos(\theta_1-\theta_2) - L\dot{\theta}_1^{\,2}\sin(\theta_1-\theta_2) = -g\sin\theta_2$$

Double pendulum

$$2\ddot{\theta}_1 + \ddot{\theta}_2 \cos(\theta_1 - \theta_2) + \dot{\theta}_2^{\,2} \sin(\theta_1 - \theta_2) + 2\frac{g}{L}\sin\theta_1 = 0$$

$$\ddot{\theta}_2 + \ddot{\theta}_1 \cos(\theta_1 - \theta_2) - \dot{\theta}_1^{\,2} \sin(\theta_1 - \theta_2) + \frac{g}{L}\sin\theta_2 = 0$$

Define:

$$\dot{\theta}_1 = \omega_1$$

$$\dot{\theta}_2 = \omega_2$$

$$2\dot{\omega}_1 + \dot{\omega}_2 \cos(\theta_1 - \theta_2) + \omega_2^2 \sin(\theta_1 - \theta_2) + 2\frac{g}{L}\sin\theta_1 = 0$$

$$\dot{\omega}_2 + \dot{\omega}_1 \cos(\theta_1 - \theta_2) - \omega_1^2 \sin(\theta_1 - \theta_2) + \frac{g}{L}\sin\theta_2 = 0$$

Double pendulum

$$2\dot{\omega}_1 + \dot{\omega}_2 \cos(\theta_1 - \theta_2) + \omega_2^2 \sin(\theta_1 - \theta_2) + 2\frac{g}{L}\sin\theta_1 = 0$$

$$\dot{\omega}_2 + \dot{\omega}_1 \cos(\theta_1 - \theta_2) - \omega_1^2 \sin(\theta_1 - \theta_2) + \frac{g}{L}\sin\theta_2 = 0$$

$$\dot{\omega}_1 = -\frac{1}{2}\dot{\omega}_2 \cos(\theta_1 - \theta_2) - \frac{1}{2}\omega_2^2 \sin(\theta_1 - \theta_2) - \frac{g}{L}\sin\theta_1$$

$$\dot{\omega}_1 \cos(\theta_1 - \theta_2) = -\dot{\omega}_2 + \omega_1^2 \sin(\theta_1 - \theta_2) - \frac{g}{L}\sin\theta_2$$

$$\dot{\omega}_1 = -\frac{\dot{\omega}_2}{\cos(\theta_1 - \theta_2)} + \omega_1^2 \frac{\sin(\theta_1 - \theta_2)}{\cos(\theta_1 - \theta_2)} - \frac{g}{L}\frac{\sin\theta_2}{\cos(\theta_1 - \theta_2)}$$

Double pendulum

$$\dot{\omega_1} = -\frac{1}{2}\dot{\omega_2}\cos(\theta_1 - \theta_2) - \frac{1}{2}\omega_2^2\sin(\theta_1 - \theta_2) - \frac{g}{L}\sin\theta_1$$

$$\dot{\omega_1} = -\frac{\dot{\omega_2}}{\cos(\theta_1 - \theta_2)} + \omega_1^2\frac{\sin(\theta_1 - \theta_2)}{\cos(\theta_1 - \theta_2)} - \frac{g}{L}\frac{\sin\theta_2}{\cos(\theta_1 - \theta_2)}$$

$$-\frac{1}{2}\dot{\omega_2}\cos(\theta_1 - \theta_2) - \frac{1}{2}\omega_2^2\sin(\theta_1 - \theta_2) - \frac{g}{L}\sin\theta_1 = -\frac{\dot{\omega_2}}{\cos(\theta_1 - \theta_2)} + \omega_1^2\frac{\sin(\theta_1 - \theta_2)}{\cos(\theta_1 - \theta_2)} - \frac{g}{L}\frac{\sin\theta_2}{\cos(\theta_1 - \theta_2)}$$

$$-\frac{1}{2}\dot{\omega_2}\cos^2(\theta_1 - \theta_2) - \frac{1}{2}\omega_2^2\sin(\theta_1 - \theta_2)\cos(\theta_1 - \theta_2) - \frac{g}{L}\sin\theta_1\cos(\theta_1 - \theta_2) = -\dot{\omega_2} + \omega_1^2\sin(\theta_1 - \theta_2) - \frac{g}{L}\sin\theta_2$$

$$\dot{\omega_2}(1 - \frac{1}{2}\cos^2(\theta_1 - \theta_2)) = \frac{1}{2}\omega_2^2\sin(\theta_1 - \theta_2)\cos(\theta_1 - \theta_2) + \frac{g}{L}\sin\theta_1\cos(\theta_1 - \theta_2) + \omega_1^2\sin(\theta_1 - \theta_2) - \frac{g}{L}\sin\theta_2$$

Double pendulum

$$\dot{\omega}_2(1-\frac{1}{2}\cos^2(\theta_1-\theta_2)) = \frac{1}{2}\omega_2^2\sin(\theta_1-\theta_2)\cos(\theta_1-\theta_2) + \frac{g}{L}\sin\theta_1\cos(\theta_1-\theta_2) + \omega_1^2\sin(\theta_1-\theta_2) - \frac{g}{L}\sin\theta_2$$

Product to sum rule: sinA cosB = 0.5[sin(A+B)+sin(A-B)]

A = B, sinA cosA = 0.5[sin(2A)+sin 0] = 0.5*sin(2A)

$$\dot{\omega}_2(1-\frac{1}{2}\cos^2(\theta_1-\theta_2)) = \frac{1}{4}\omega_2^2\sin(2\theta_1-2\theta_2) + \frac{g}{L}\sin\theta_1\cos(\theta_1-\theta_2) + \omega_1^2\sin(\theta_1-\theta_2) - \frac{g}{L}\sin\theta_2$$

Product to sum rule: sinA cosB = 0.5[sin(A+B)+sin(A-B)]

$A = \theta_1$, $B = \theta_1 - \theta_2$, $0.5[\sin 2\theta_1 - \theta_2 + \sin \theta_2]$

$$\dot{\omega}_2(1-\frac{1}{2}\cos^2(\theta_1-\theta_2)) = \frac{1}{4}\omega_2^2\sin(2\theta_1-2\theta_2) + \frac{g}{2L}(\sin(2\theta_1-\theta_2)+\sin\theta_2) + \omega_1^2\sin(\theta_1-\theta_2) - \frac{g}{L}\sin\theta_2$$

$$\dot{\omega}_2(1-\frac{1}{2}\cos^2(\theta_1-\theta_2)) = \frac{1}{4}\omega_2^2\sin(2\theta_1-2\theta_2) + \frac{g}{2L}(\sin(2\theta_1-\theta_2)-\sin\theta_2) + \omega_1^2\sin(\theta_1-\theta_2)$$

Double pendulum

$$\dot{\omega}_2\left(1 - \frac{1}{2}\cos^2(\theta_1 - \theta_2)\right) = \frac{1}{4}\omega_2^2 \sin(2\theta_1 - 2\theta_2) + \frac{g}{2L}\left(\sin(2\theta_1 - \theta_2) - \sin\theta_2\right) + \omega_1^2 \sin(\theta_1 - \theta_2)$$

$$\boxed{\dot{\omega}_2 = \frac{1}{1 - \frac{1}{2}\cos^2(\theta_1 - \theta_2)}\left[\frac{1}{4}\omega_2^2 \sin(2\theta_1 - 2\theta_2) + \frac{g}{2L}\left(\sin(2\theta_1 - \theta_2) - \sin\theta_2\right) + \omega_1^2 \sin(\theta_1 - \theta_2)\right]}$$

From before:

$$2\dot{\omega}_1 + \dot{\omega}_2 \cos(\theta_1 - \theta_2) + \omega_2^2 \sin(\theta_1 - \theta_2) + 2\frac{g}{L}\sin\theta_1 = 0$$

$$\dot{\omega}_2 + \dot{\omega}_1 \cos(\theta_1 - \theta_2) - \omega_1^2 \sin(\theta_1 - \theta_2) + \frac{g}{L}\sin\theta_2 = 0$$

$$\dot{\omega}_2 \cos(\theta_1 - \theta_2) = -2\dot{\omega}_1 - \omega_2^2 \sin(\theta_1 - \theta_2) - 2\frac{g}{L}\sin\theta_1$$

$$\dot{\omega}_2 = -\dot{\omega}_1 \cos(\theta_1 - \theta_2) + \omega_1^2 \sin(\theta_1 - \theta_2) - \frac{g}{L}\sin\theta_2$$

Double pendulum

$$\dot{\omega}_2 \cos(\theta_1 - \theta_2) = -2\dot{\omega}_1 - \omega_2^2 \sin(\theta_1 - \theta_2) - 2\frac{g}{L}\sin\theta_1$$

$$\dot{\omega}_2 = -\dot{\omega}_1 \cos(\theta_1 - \theta_2) + \omega_1^2 \sin(\theta_1 - \theta_2) - \frac{g}{L}\sin\theta_2$$

$$\dot{\omega}_2 = \frac{-2\dot{\omega}_1}{\cos(\theta_1 - \theta_2)} - \frac{\omega_2^2 \sin(\theta_1 - \theta_2)}{\cos(\theta_1 - \theta_2)} - \frac{2\frac{g}{L}\sin\theta_1}{\cos(\theta_1 - \theta_2)}$$

$$\frac{-2\dot{\omega}_1}{\cos(\theta_1 - \theta_2)} - \frac{\omega_2^2 \sin(\theta_1 - \theta_2)}{\cos(\theta_1 - \theta_2)} - \frac{2\frac{g}{L}\sin\theta_1}{\cos(\theta_1 - \theta_2)} =$$

$$-\dot{\omega}_1 \cos(\theta_1 - \theta_2) + \omega_1^2 \sin(\theta_1 - \theta_2) - \frac{g}{L}\sin\theta_2$$

Double pendulum

$$\frac{-2\dot{\omega}_1}{\cos(\theta_1-\theta_2)} - \frac{\omega_2^2\sin(\theta_1-\theta_2)}{\cos(\theta_1-\theta_2)} - \frac{2\frac{g}{L}\sin\theta_1}{\cos(\theta_1-\theta_2)} =$$

$$-\dot{\omega}_1\cos(\theta_1-\theta_2) + \omega_1^2\sin(\theta_1-\theta_2) - \frac{g}{L}\sin\theta_2$$

$$-2\dot{\omega}_1 - \omega_2^2\sin(\theta_1-\theta_2) - 2\frac{g}{L}\sin\theta_1 =$$

$$-\dot{\omega}_1\cos^2(\theta_1-\theta_2) + \omega_1^2\sin(\theta_1-\theta_2)\cos(\theta_1-\theta_2) - \frac{g}{L}\sin\theta_2\cos(\theta_1-\theta_2)$$

$$\dot{\omega}_1(-2+\cos^2(\theta_1-\theta_2)) = \omega_2^2\sin(\theta_1-\theta_2) + \frac{g}{L}(2\sin\theta_1 - \sin\theta_2\cos(\theta_1-\theta_2)) + \omega_1^2\sin(\theta_1-\theta_2)\cos(\theta_1-\theta_2)$$

$$\dot{\omega}_1(-2+\cos^2(\theta_1-\theta_2)) = \omega_2^2\sin(\theta_1-\theta_2) + \frac{g}{L}(2\sin\theta_1 - \sin\theta_2\cos(\theta_1-\theta_2)) + \frac{1}{2}\omega_1^2\sin(2\theta_1-2\theta_2)$$

Double pendulum

$$\dot{\omega}_1(-2+\cos^2(\theta_1-\theta_2)) = \omega_2^2\sin(\theta_1-\theta_2) + \frac{g}{L}(2\sin\theta_1 - \sin\theta_2\cos(\theta_1-\theta_2)) + \frac{1}{2}\omega_1^2\sin(2\theta_1-2\theta_2)$$

Product to sum rule: sinA cosB = 0.5[sin(A+B)+sin(A-B)]

$A = \theta_2$, $B = \theta_1-\theta_2$, $0.5[\sin\theta_1 + \sin 2\theta_2-\theta_1] =$

$$\dot{\omega}_1(-2+\cos^2(\theta_1-\theta_2)) = \omega_2^2\sin(\theta_1-\theta_2) + \frac{g}{L}(2\sin\theta_1 - \frac{1}{2}\sin\theta_1 - \frac{1}{2}\sin(2\theta_2-\theta_1)) + \frac{1}{2}\omega_1^2\sin(2\theta_1-2\theta_2)$$

$$\dot{\omega}_1(-2+\cos^2(\theta_1-\theta_2)) = \omega_2^2\sin(\theta_1-\theta_2) + \frac{g}{L}(\frac{3}{2}\sin\theta_1 + \frac{1}{2}\sin(2\theta_2-\theta_1)) + \frac{1}{2}\omega_1^2\sin(2\theta_1-2\theta_2)$$

$$\dot{\omega}_1(-2+\cos^2(\theta_1-\theta_2)) = \omega_2^2\sin(\theta_1-\theta_2) + \frac{g}{L}(\frac{3}{2}\sin\theta_1 + \frac{1}{2}\sin(\theta_1-2\theta_2)) + \frac{1}{2}\omega_1^2\sin(2\theta_1-2\theta_2)$$

$$\dot{\omega}_1 = \frac{-1}{(2-\cos^2(\theta_1-\theta_2))}\left[\omega_2^2\sin(\theta_1-\theta_2) + \frac{g}{L}(\frac{3}{2}\sin\theta_1 + \frac{1}{2}\sin(\theta_1-2\theta_2)) + \frac{1}{2}\omega_1^2\sin(2\theta_1-2\theta_2)\right]$$

$$\dot{\omega}_1 = \frac{-0.5}{(1-\frac{1}{2}\cos^2(\theta_1-\theta_2))}\left[\omega_2^2\sin(\theta_1-\theta_2) + \frac{g}{L}(\frac{3}{2}\sin\theta_1 + \frac{1}{2}\sin(\theta_1-2\theta_2)) + \frac{1}{2}\omega_1^2\sin(2\theta_1-2\theta_2)\right]$$

Double pendulum

$$\dot{\omega}_1 = \frac{-0.5}{(1 - \frac{1}{2}\cos^2(\theta_1 - \theta_2))}\left[\omega_2^2 \sin(\theta_1 - \theta_2) + \frac{g}{L}(\frac{3}{2}\sin\theta_1 + \frac{1}{2}\sin(\theta_1 - 2\theta_2)) + \frac{1}{2}\omega_1^2 \sin(2\theta_1 - 2\theta_2)\right]$$

$$\dot{\omega}_2 = \frac{1}{1 - \frac{1}{2}\cos^2(\theta_1 - \theta_2)}\left[\frac{1}{4}\omega_2^2 \sin(2\theta_1 - 2\theta_2) + \frac{g}{2L}\left(\sin(2\theta_1 - \theta_2) - \sin\theta_2\right) + \omega_1^2 \sin(\theta_1 - \theta_2)\right]$$

PHEW! Slightly different form from the book, but they are equivalent

Double pendulum with vpython


```
from math import sin,cos,pi
from numpy import arange,array
from vpython import rate, vector, sphere,cylinder

g = 9.81
l = 0.4
R = 0.05
W = 0.01
framerate = 200
stepsperframe = 20

# Function f(r)
def f(r):
  theta1 = r[0]
  theta2 = r[1]
  omega1 = r[2]
  omega2 = r[3]
  ftheta1 = omega1
  ftheta2 = omega2
  fomega1 = -(omega1*omega1*sin(2*theta1-2*theta2) + 2*omega2*omega2*sin(theta1-theta2) + (3*sin(theta1) + sin(theta1-2*theta2))*g/l)/(3-cos(2*theta1-2*theta2))
  fomega2 = (4*omega1*omega1*sin(theta1-theta2) + omega2*omega2*sin(2*theta1-2*theta2) - 2*(sin(theta2)-sin(2*theta1-theta2))*g/l)/(3-cos(2*theta1-2*theta2))
  return array([ftheta1,ftheta2,fomega1,fomega2],float)
```

Double pendulum with vpython


```
# Main Program
# Set up starting values
h = 1.0/(framerate*stepsperframe)
theta1 = 0.8*pi
theta2 = 0.9*pi
omega1 = omega2 = 0.0
#theta1 = theta2 = 0.5*pi
#omega1 = omega2 = 0.0
r = array([theta1,theta2,omega1,omega2],float)

# Set up graphics
pivot = sphere(pos=vector(0,0,0),radius=R)
x1 = l*sin(theta1)
y1 = -l*cos(theta1)
x2 = l*(sin(theta1)+sin(theta2))
y2 = -l*(cos(theta1)+cos(theta2))
arm1 = cylinder(pos = vector(0,0,0),axis=vector(x1,y1,0),radius=W)
arm2 = cylinder(pos = vector(x1,y1,0), axis=vector(x2-x1,y2-y1,0), radius=W)
bob1 = sphere(pos = vector(x1,y1,0),radius=R)
bob2 = sphere(pos = vector(x2,y2,0),radius=R)
```

Double pendulum with vpython


```
# Main loop
while True:
   for i in range(stepsperframe):
      k1 = h*f(r)
      k2 = h*f(r+0.5*k1)
      k3 = h*f(r+0.5*k2)
      k4 = h*f(r+k3)
      r += (k1+2*k2+2*k3+k4)/6

      # Graphics
      theta1 = r[0]
      theta2 = r[1]
      x1 = l*sin(theta1)
      y1 = -l*cos(theta1)
      x2 = l*(sin(theta1)+sin(theta2))
      y2 = -l*(cos(theta1)+cos(theta2))
      rate(framerate)
      arm1.axis = vector(x1,y1,0)
      arm2.pos = vector(x1,y1,0)
      arm2.axis = vector(x2-x1,y2-y1,0)
      bob1.pos = vector(x1,y1,0)
      bob2.pos = vector(x2,y2,0)
```

Runs indefinitely! Let's see if my poor old laptop can execute the program, and let's look at the results

Investigating double pendulum some more

```python
# Double pendulum, some interesting plots (not animation)
from math import sin,cos,pi
from numpy import arange,array

# Function f(r)
def f(r,l):
  g = 9.81
  theta1 = r[0]
  theta2 = r[1]
  omega1 = r[2]
  omega2 = r[3]
  ftheta1 = omega1
  ftheta2 = omega2
  fomega1 = -(omega1*omega1*sin(2*theta1-2*theta2) + 2*omega2*omega2*sin(theta1-theta2) + (3*sin(theta1) + sin(theta1-2*theta2))*g/l)/(3-cos(2*theta1-2*theta2))
  fomega2 = (4*omega1*omega1*sin(theta1-theta2) + omega2*omega2*sin(2*theta1-2*theta2) - 2*(sin(theta2)-sin(2*theta1-theta2))*g/l)/(3-cos(2*theta1-2*theta2))
  return array([ftheta1,ftheta2,fomega1,fomega2],float)
```

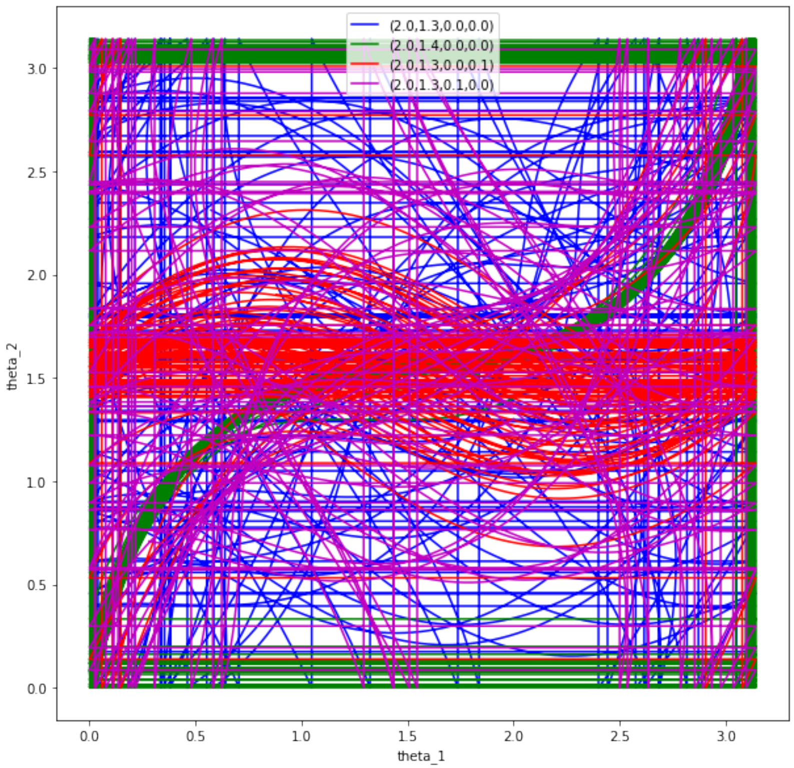
```python
# Main Program from arbitrary starting values
def runDoublePendulum(theta1,theta2,omega1,omega2,tf=5.,nstep=3000000,l = 1.0):
  t = 0
  h = (tf-t)/nstep
  r = array([theta1,theta2,omega1,omega2])
  thetas_1 = []
  thetas_2 = []
  omegas_1 = []
  omegas_2 = []

  # Main loop
  while t < tf:
    ### fix values so we're not sensitive to 2pi wrap
    while (r[0] < pi): r[0] += pi
    while (r[1] < pi): r[1] += pi
    while (r[0] > pi): r[0] -= pi
    while (r[1] > pi): r[1] -= pi
    thetas_1.append(r[0])
    thetas_2.append(r[1])
    omegas_1.append(r[2])
    omegas_2.append(r[3])

    k1 = h*f(r,l)
    k2 = h*f(r+0.5*k1,l)
    k3 = h*f(r+0.5*k2,l)
    k4 = h*f(r+k3,l)
    r += (k1+2*k2+2*k3+k4)/6
```

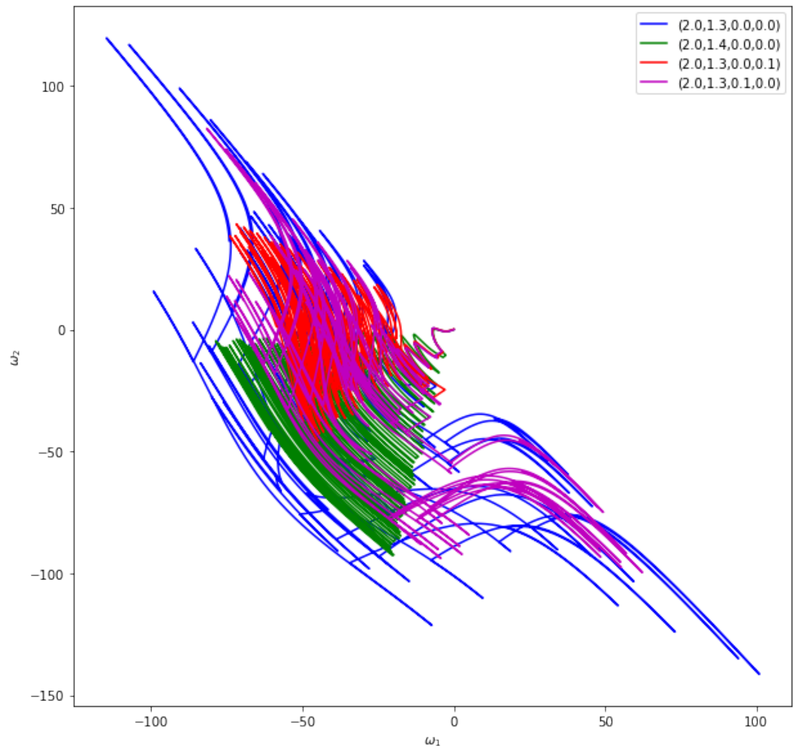
Make code reusable and easily configurable, be careful about 2pi effects!

Investigating double pendulum some more


```
from matplotlib.pyplot import plot,show,figure,xlabel,ylabel,legend

thetas_1_a, thetas_2_a, omegas_1_a, omegas_2_a = runDoublePendulum(2.0,1.3,0.0,0.0)
thetas_1_b, thetas_2_b, omegas_1_b, omegas_2_b = runDoublePendulum(2.0,1.4,0.0,0.0)
thetas_1_c, thetas_2_c, omegas_1_c, omegas_2_c = runDoublePendulum(2.0,1.3,0.0,0.1)
thetas_1_d, thetas_2_d, omegas_1_d, omegas_2_d = runDoublePendulum(2.0,1.3,0.1,0.0)
fig = figure(figsize=(10,10))
plot(thetas_1_a,thetas_2_a,label="(2.0,1.3,0.0,0.0)",color='b')
plot(thetas_1_b,thetas_2_b,label="(2.0,1.4,0.0,0.0)",color='g')
plot(thetas_1_c,thetas_2_c,label="(2.0,1.3,0.0,0.1)",color='r')
plot(thetas_1_d,thetas_2_d,label="(2.0,1.3,0.1,0.0)",color='m')
xlabel("theta_1")
ylabel("theta_2")
legend()
show()

fig = figure(figsize=(10,10))
plot(omegas_1_a,omegas_2_a,label="(2.0,1.3,0.0,0.0)",color='b')
plot(omegas_1_b,omegas_2_b,label="(2.0,1.4,0.0,0.0)",color='g')
plot(omegas_1_c,omegas_2_c,label="(2.0,1.3,0.0,0.1)",color='r')
plot(omegas_1_d,omegas_2_d,label="(2.0,1.3,0.1,0.0)",color='m')
xlabel("$\omega_1$")
ylabel("$\omega_2$")
legend()
show()
```

Investigating double pendulum some more





Small changes in initial conditions lead to very large changes later on!

Double pendulum with matplotlib


```
from math import sin,cos,pi
from numpy import arange,array
import matplotlib.pyplot as plt
import matplotlib.animation as animation

g = 9.81
l = 0.4
R = 0.05
W = 0.01
stepsperframe = 20
framerate = 200
```

```
# Main Program
# Set up starting values
h = 1.0/(framerate*stepsperframe)
theta1 = 0.8*pi
theta2 = 0.9*pi
omega1 = omega2 = 0.0
r = array([theta1,theta2,omega1,omega2],float)

def init():
    x1 = l*sin(theta1)
    y1 = -l*cos(theta1)
    x2 = l*(sin(theta1)+sin(theta2))
    y2 = -l*(cos(theta1)+cos(theta2))
    line.set_data([], [])
    return line
```

Advantage here: we can save our animation!
Disadvantage: Syntax is awkward, and it's slow

Double pendulum with matplotlib

```
def animate(i):
    global r
    for i in range(stepsperframe):
        k1 = h*f(r)
        k2 = h*f(r+0.5*k1)
        k3 = h*f(r+0.5*k2)
        k4 = h*f(r+k3)
        r += (k1+2*k2+2*k3+k4)/6

        # Graphics
        theta1 = r[0]
        theta2 = r[1]
        x1 = l*sin(theta1)
        y1 = -l*cos(theta1)
        x2 = l*(sin(theta1)+sin(theta2))
        y2 = -l*(cos(theta1)+cos(theta2))
        thisx = [0, x1, x2]
        thisy = [0, y1, y2]
        line.set_data(thisx, thisy)
    return line,
```

Idea: Define a function that steps through the animation one-by-one and returns the objects to be animated. Let's open the output in the web browser

```
fig = plt.figure()
ax = plt.axes(xlim=(-2*l, 2*l), ylim=(-2*l,2*l))
line, = ax.plot([], [], lw=3)
ani = animation.FuncAnimation(fig, animate, interval=framerate, frames = 5000, init_func=init)
###plt.show()
ani.save('double_pendulum.mp4', fps=60, extra_args=['-vcodec', 'libx264'])
```

Double pendulum with matplotlib

```
### Double pendulum animation
from math import sin,cos,pi
from numpy import arange,array
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from matplotlib import rc

from IPython.display import HTML
```

Can also do this in colab if we use a few extra tricks (and don't make the animation too big)

```
fig = plt.figure()
ax = plt.axes(xlim=(-2*l, 2*l), ylim=(-2*l,2*l))
line, = ax.plot([], [], lw=3)
ani = animation.FuncAnimation(fig, animate, interval=framerate, frames = 2000, init_func=init)

# Note: below is the part which makes it work on Colab
rc('animation', html='jshtml')
ani
```

Single non-linear pendulum


```python
### Exercise 8.4 Non-linear pendulum, animation

from math import sin,cos,pi
from numpy import array
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from matplotlib import rc

from IPython.display import HTML

g = 9.81
l = 0.1
W = 0.002
R = 0.01
framerate = 20
stepsperframe = 50

def f(r):
  theta = r[0]
  omega = r[1]
  ftheta = omega
  fomega = (-g/l)*sin(theta)
  return array([ftheta,fomega],float)

# Initial values
theta = pi*179/180
r = array([theta,0.0],float)
# Main loop
h = 1.0/(framerate*stepsperframe)
```

```python
def init():
    x = l*sin(theta)
    y = -l*cos(theta)
    line.set_data([], [])
    return line

def animate(i):
  global r
  for i in range(stepsperframe):
    k1 = h*f(r)
    k2 = h*f(r+0.5*k1)
    k3 = h*f(r+0.5*k2)
    k4 = h*f(r+k3)
    r += (k1+2*k2+2*k3+k4)/6
    # Graphics
    theta = r[0]
    x = [0,l*sin(theta),0]
    y = [0,-l*cos(theta),0]
    line.set_data(x,y)
    return line,

fig = plt.figure()
ax = plt.axes(xlim=(-2*l, 2*l), ylim=(-2*l,2*l))
line, = ax.plot([], [], lw=3)
ani = animation.FuncAnimation(fig, animate, interval=framerate, frames = 20000, init_func=init)

# Note: below is the part which makes it work on Colab
rc('animation', html='jshtml')
ani
```

Homework #6

8.1, 8.2, 8.6, 8.12

Also 8.7 (PHYS 510 only)