Kai-Feng Chen
National Taiwan University

PROGRAMMING & NUMERICAL ANALYSIS

2016

Lecture 11: Solving Differential Equations

WORK OF "PHYSICISTS"

- Solving the differential equations is probably one of your most "ordinary" work when you study the classical mechanics?
- Many differential equations in nature cannot be solved analytically easily; however, in many of the cases, a numeric approximation to the solution is often good enough to solve the problem. You will see several examples today.
- In this lecture we will discuss the numerical methods for finding numerical approximations to the solutions of ordinary differential equations, as well as how to demonstrate the "motions" with an animation in matplotlib.

WORK OF "PHYSICISTS" (II)

- Let's get back to our "lovely" **F=ma** equations!

THE BASIS: A BRAINLESS EXAMPLE

- Let's try to solve such a (mostly) trivial differential equation:

$$\frac{dy}{dt} = f(y,t) = y$$

with the initial condition:
t = 0, y = 1

- You should know the obvious solution is — $\boldsymbol{y = \exp(t)}$

$$\frac{dy}{dt} = f(y,t)$$

Actually, this is the **general form** of any first-order ordinary differential equation.

In general, it can be very complicated, but it's still a 1st order ODE, e.g.

$$\frac{dy}{dt} = f(y,t) = y^3 \cdot t^2 + \sin(t+y) + \sqrt{t+y}$$

THE NUMERICAL SOLUTION

- Here are the minimal algorithm — integrate the differential equation by **one step in t**:

$$\frac{dy}{dt} = f(y,t)$$

$$\frac{y(t_{n+1}) - y(t_n)}{h} = f(y,t_n) \quad \Longrightarrow \quad y_{n+1} \approx y_n + h \cdot f(y_n, t_n)$$

next step **current step**

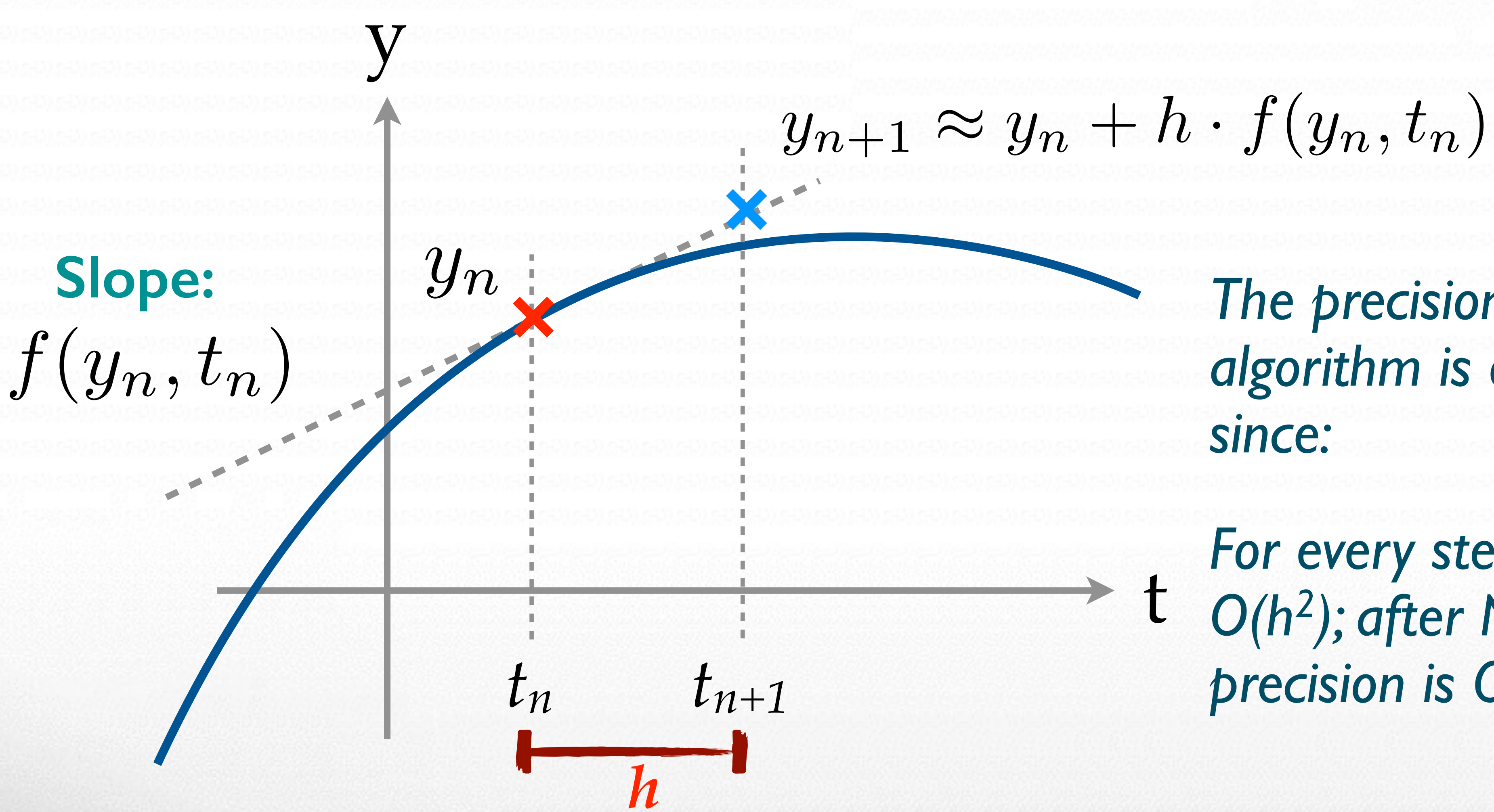
For our trivial example:

$$\frac{dy}{dt} = y \quad \Longrightarrow \quad y_{n+1} \approx y_n + h \cdot y_n$$

This is the classical **Euler algorithm (method)**

EULER ALGORITHM

- A more graphical explanation is as like this:



The precision of this Euler algorithm is only up to O(h) since:

For every step the precision is of $O(h^2)$; after N~O(1/h) steps the precision is O(h).

EULER ALGORITHM (II)

- Let's prepare a simple code to see how it works:

```
import math

def f(t,y): return y

t, y = 0., 1.  ⇐ Initial conditions (t = 0, y = 1)
h = 0.001  ⇐ stepping in t

while t<1.:
    k1  = f(t, y)  ⇐ the given f(y,t) function
    y  += h*k1
    t  += h

y_exact = math.exp(t)
print 'Euler method: %.16f, exact: %.16f, diff: %.16f' % \
(y,y_exact,abs(y-y_exact))
```

111-example-01.py

```
Euler method: 2.7169239322358960,
exact:        2.7182818284590469,
diff:         0.0013578962231509
```

⇐ Indeed the precision is of O(h)

SECOND ORDER RUNGE-KUTTA METHOD

- Surely one can introduce a similar trick of error reduction we have played though out the latter half of the semester.
- Here comes the **Runge-Kutta algorithm** for integrating differential equations, which is based on a formal integration:

$$\frac{dy}{dt} = f(y,t) \quad \Rightarrow \quad y(t) = \int f(t,y)dt$$

$$y_{n+1} = y_n + \int_{t_n}^{t_{n+1}} f(t,y)dt$$

Expand ***f(t,y)*** in a Taylor series around $(t,y) = (t_{n+\frac{1}{2}}, y_{n+\frac{1}{2}})$

$$f(t,y) = f(t_{n+\frac{1}{2}}, y_{n+\frac{1}{2}}) + (t - t_{n+\frac{1}{2}}) \cdot \frac{df}{dt}(t_{n+\frac{1}{2}}) + O(h^2)$$

Something smells familiar?

SECOND ORDER RUNGE-KUTTA METHOD (II)

$$f(t,y) = f(t_{n+\frac{1}{2}}, y_{n+\frac{1}{2}}) + (t - t_{n+\frac{1}{2}}) \cdot \boxed{\frac{df}{dt}(t_{n+\frac{1}{2}})} + O(h^2)$$

Insert the expansion into the integration:

It's just a number (slope)!

$$\int_{t_n}^{t_{n+1}} f(t,y)dt = \int_{t_n}^{t_{n+1}} f(t_{n+\frac{1}{2}}, y_{n+\frac{1}{2}})dt + \int_{t_n}^{t_{n+1}} (t - t_{n+\frac{1}{2}}) \cdot \boxed{\frac{df}{dt}(t_{n+\frac{1}{2}})}\, dt + \ldots$$

Linear (first order) term must be cancelled

Insert the integral back:

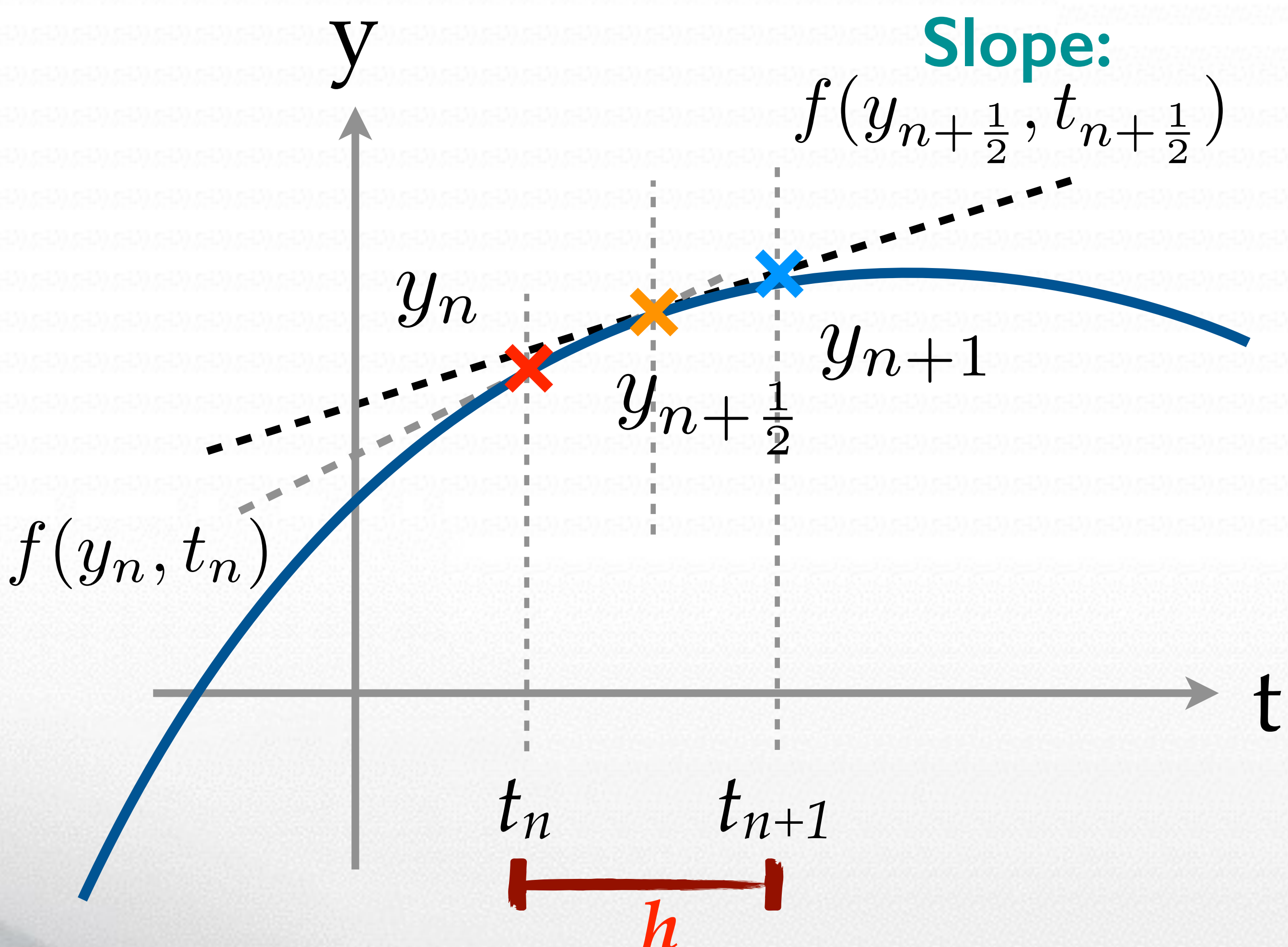
$$\int_{t_n}^{t_{n+1}} f(t,y)dt \approx h \cdot f(t_{n+\frac{1}{2}}, y_{n+\frac{1}{2}})$$

$$\Rightarrow \quad y_{n+1} \approx y_n + h \cdot f(t_{n+\frac{1}{2}}, y_{n+\frac{1}{2}}) + O(h^3)$$

If one knows the solution **half-step in the future** — the $O(h^2)$ term can be cancelled. ***BUT HOW?***

SECOND ORDER RUNGE-KUTTA METHOD (III)

- The trick: use the **Euler's method to solve half-step first**, starting from the given initial conditions:



$$\begin{cases} y_{n+\frac{1}{2}} = y_n + \dfrac{h}{2} f(t_n, y_n) \\ t_{n+\frac{1}{2}} = t + \dfrac{h}{2} \\ y_{n+1} \approx y_n + h \cdot f(t_{n+\frac{1}{2}}, y_{n+\frac{1}{2}}) \end{cases}$$

Explicit formulae

$$\begin{cases} k_1 = f(t_n, y_n) \\ k_2 = f(t_n + \dfrac{h}{2}, y_n + \dfrac{h}{2} \cdot k_1) \\ y_{n+1} \approx y_n + h \cdot k_2 + O(h^3) \end{cases}$$

IMPLEMENTATION OF "RK2"

- The coding is actually extremely simple:

```
t, y = 0., 1.      ⇐ Initial conditions and stepping (t = 0, y = 1, h = 0.001)
h = 0.001

while t<1.:
    k1  = f(t, y)     ⇐ use Euler method to solve half-step
    k2  = f(t+0.5*h, y+0.5*h*k1)
    y  += h*k2    ⇐ full step jump                              RK2 solver
    t  += h

y_exact = math.exp(t)
print 'RK2 method: %.16f, exact: %.16f, diff: %.16f' % \
(y,y_exact,abs(y-y_exact))
```

111-example-02.py

```
RK2 method:    2.7182813757517628,
exact:         2.7182818284590469,
diff:          0.0000004527072841
```

For every step the precision is of $O(h^3)$; after N steps the precision is $O(h^2)$.

FOURTH ORDER RUNGE-KUTTA

- The **4th order Runge-Kutta method** provides an excellent balance of power, precision, and programming simplicity. Using a similar idea of the 2nd order version, one could have these formulae:

$$
\begin{cases}
k_1 = f(t_n, y_n) \\
k_2 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2} \cdot k_1) \\
k_3 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2} \cdot k_2) \\
k_4 = f(t_n + h, y_n + h \cdot k_3)
\end{cases}
$$

$$y_{n+1} \approx y_n + \frac{h}{6} \cdot (k_1 + 2k_2 + 2k_3 + k_4) + O(h^5)$$

Basically the 4th order Runge-Kutta has a precision of $O(h^5)$ at each step, an over all **$O(h^4)$** precision.

Actually, the RK4 is a variation of **Simpson's method**...

IMPLEMENTATION OF "RK4"

- The RK4 routine is not too different from the previous RK2 code!

```
t, y = 0., 1.
h = 0.001

while t<1.:
    k1  = f(t, y)
    k2  = f(t+0.5*h, y+0.5*h*k1)
    k3  = f(t+0.5*h, y+0.5*h*k2)
    k4  = f(t+h, y+h*k3)
    y  += h/6.*(k1+2.*k2+2.*k3+k4)
    t  += h

y_exact = math.exp(t)
print 'RK2 method: %.16f, exact: %.16f, diff: %.16f' % \
(y,y_exact,abs(y-y_exact))
```

⇐ The same initial conditions & stepping

RK4 solver

⇐ Simply calculate k1~k4 in a sequence

⇐ Jump to the next step

111-example-03.py

```
RK4 method:    2.7182818284590247,
exact:         2.7182818284590469,
diff:          0.0000000000000222
```

⇐ Precision is of $O(h^4)$!

PRECISION EVOLUTION

- Let's write a small code to demonstrate the "precision" of the solution as it evolves.
- You should be able to see the "accumulation" of numerical errors.

```
vt = np.zeros(200)
vy = np.zeros((4,200))

t = 0.
y1 = y2 = y4 = 1.
h = 0.001

for idx in range(200):
    for step in range(1000):
        k1  = f(t, y1)
        y1 += h*k1

        k1  = f(t, y2)
        k2  = f(t+0.5*h, y2+0.5*h*k1)
        y2 += h*k2

        k1  = f(t, y4)
        k2  = f(t+0.5*h, y4+0.5*h*k1)
        k3  = f(t+0.5*h, y4+0.5*h*k2)
        k4  = f(t+h, y4+h*k3)
        y4 += h/6.*(k1+2.*k2+2.*k3+k4)

        t += h

    vt[idx] = t
    vy[0,idx] = np.exp(t)
    vy[1,idx] = y1
    vy[2,idx] = y2
    vy[3,idx] = y4
```

⇐ NumPy arrays for storing the output

Only keep the result every 1000 steps.

Euler method

RK2

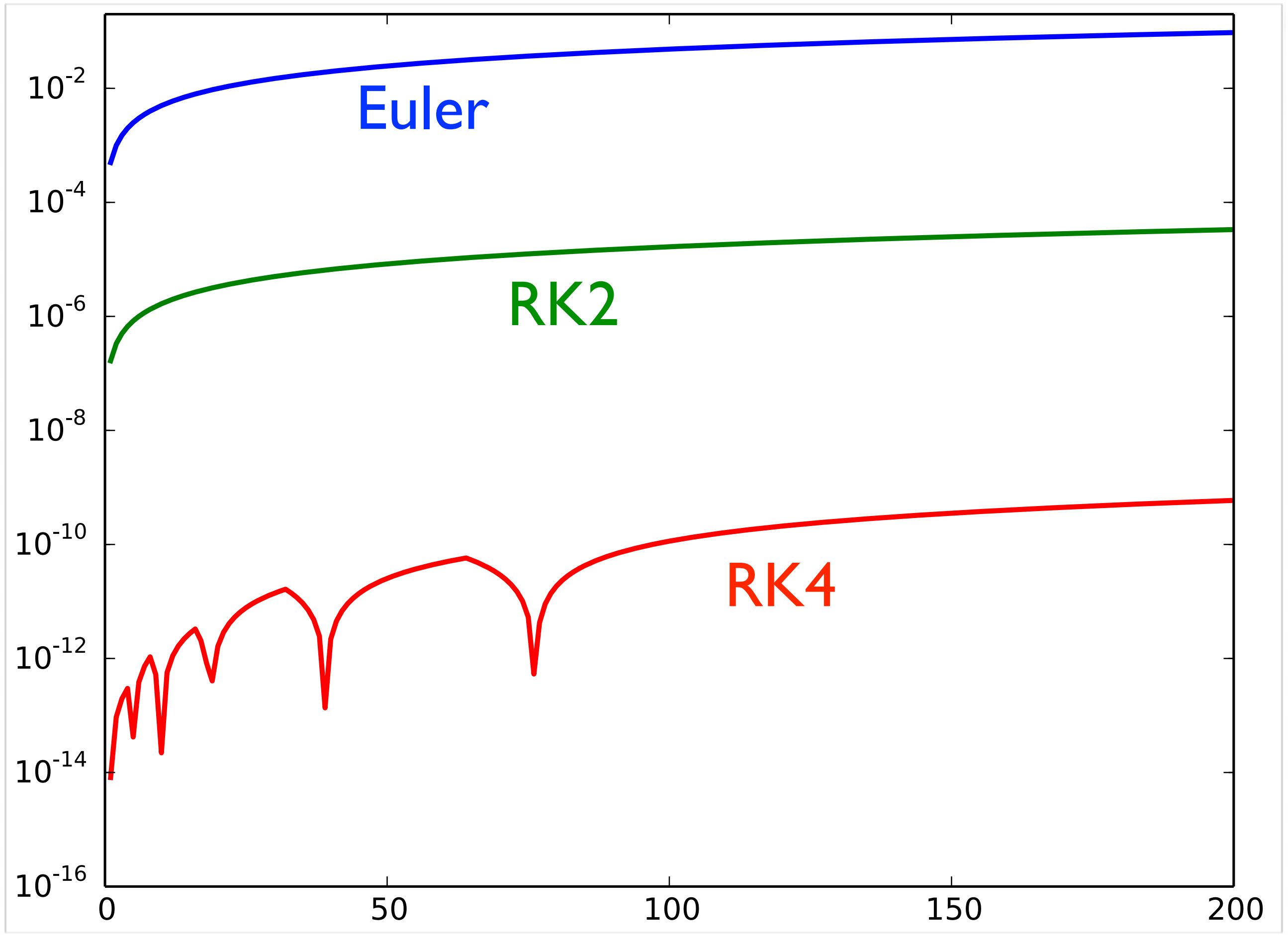
RK4

⇐ Store the results

partial 111-example-04.py

PRECISION EVOLUTION (II)

- Just make a simple plot.
- The initial uncertainties are of $O(h)$, $O(h^2)$, and $O(h^4)$.
- After 200,000 steps or more, the accumulated errors can be large.




```
plt.plot(vt,abs(vy[1]-vy[0])/vy[0],lw=2,c='Blue')
plt.plot(vt,abs(vy[2]-vy[0])/vy[0],lw=2,c='Green')
plt.plot(vt,abs(vy[3]-vy[0])/vy[0],lw=2,c='Red')
plt.yscale('log')
plt.ylim(1E-16,0.2)
plt.xlim(0.,200.)
plt.show()
```

⇑ Draw the relative differences

partial 111-example-04.py

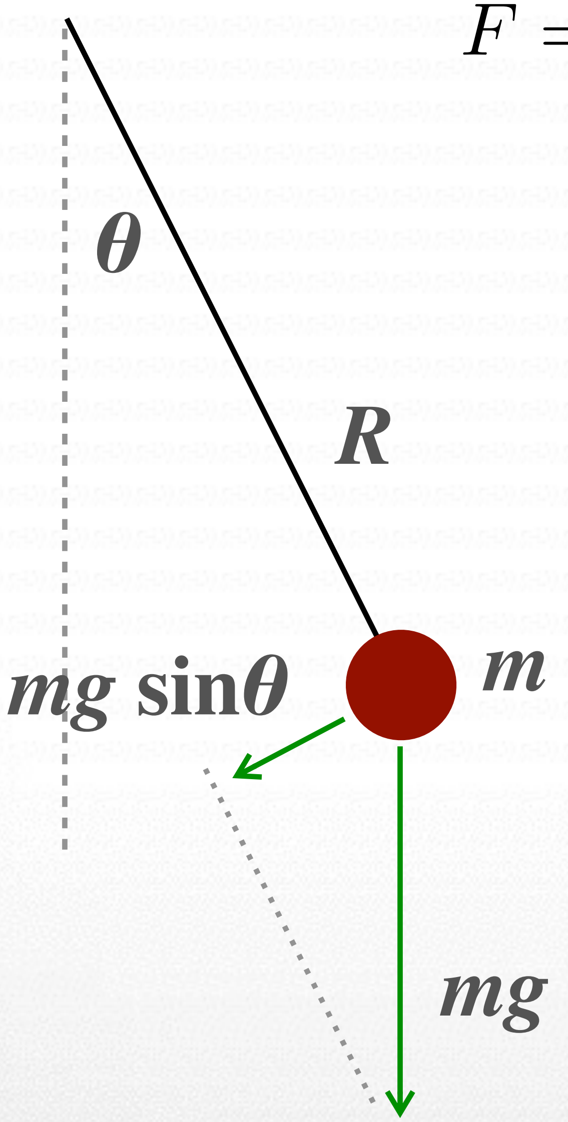
INTERMISSION

- It could be interesting to solve some other trivial differential equations with the methods introduced above, for example:

$$\frac{dy}{dt} = -y$$
$$\frac{dy}{dt} = \cos(t)$$

- Try to modify the previous example code (l11-example-04.py) and see how the error accumulated along with steps for a different differential equation.

A LITTLE BIT OF PHYSICS: SIMPLE PENDULUM



$$F = ma \quad \longrightarrow \quad mR\frac{d^2\theta}{dt^2} = -mg\sin\theta$$

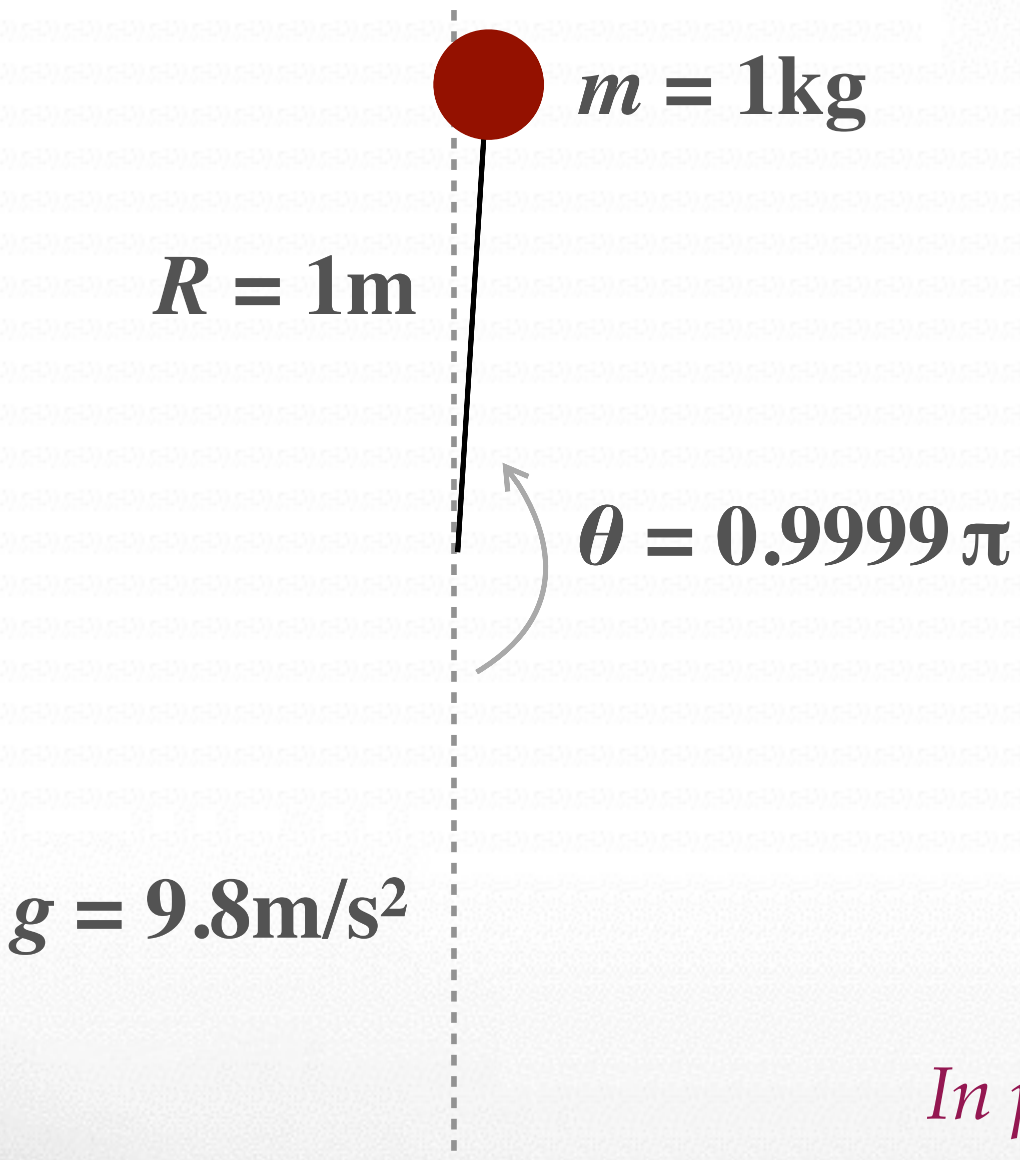
$$\frac{d^2\theta}{dt^2} = -\frac{g}{R}\sin\theta$$

Solving 2nd order ODE =
Decompose into two 1st order ODE:

$$\frac{d\dot{\theta}}{dt} = f(\theta, \dot{\theta}, t) = -\frac{g}{R}\sin\theta \quad \text{.... (1)}$$

$$\frac{d\theta}{dt} = g(\theta, \dot{\theta}, t) = \dot{\theta} \quad \text{.... (2)}$$

A LITTLE BIT OF PHYSICS: SIMPLE PENDULUM (II)



With a trial Initial condition
at **t = 0** :

$$\begin{cases} \theta = 0.9999\pi \approx 3.141278... \\ \dot{\theta} = 0 \end{cases}$$

Almost at the largest possible angle
(No small angle approximation!
Not a "simple" pendulum)
Standstill at the beginning.

In principle it should stand for a moment, and start to falling down...

SOLVE FOR 2 ODE'S TOGETHER

```
m, g, R = 1., 9.8, 1.
t, h = 0., 0.001              ⇐ Initial condition t = 0 sec, stepping = 0.001 sec.
y = np.array([np.pi*0.9999,0.])  ⇐ Initial θ and θ'

def f(t,y):
    theta    = y[0]       ⇐ input array contains θ and θ'
    thetap   = y[1]
    thetapp = -g/R*np.sin(theta)  ⇐ output array contains θ' and θ''

    return np.array([thetap,thetapp])

while t<8.:
    for step in range(100):   ⇐ solve for 100 steps (=0.1 sec)
        k1  = f(t, y)
        y  += h*k1          ⇐ Euler method
        t  += h

    theta  = y[0]
    thetap = y[1]
    print 'At %.2f sec : (%+14.10f, %+14.10f)' % (t, theta, thetap)
```

111-example-05.py

SOLVE FOR 2 ODE'S TOGETHER (II)

- The terminal output:
- Works, but not so straight forward...

Let's introduce some **animations** to demonstrate the motion!

$\theta \downarrow$ $\dot{\theta} \downarrow$

```
At 0.10 sec : ( +3.1412631358,  -0.0003127772)
At 0.20 sec : ( +3.1412152508,  -0.0006561363)
At 0.30 sec : ( +3.1411301423,  -0.0010639557)
At 0.40 sec : ( +3.1409994419,  -0.0015764466)
At 0.50 sec : ( +3.1408102869,  -0.0022441174)
... ...
At 1.00 sec : ( +3.1380085436,  -0.0111772696)
... ...
At 1.50 sec : ( +3.1245199136,  -0.0534365650)
... ...
At 2.00 sec : ( +3.0601357015,  -0.2549284063)
... ...
At 2.50 sec : ( +2.7540224966,  -1.2057243644)
... ...
At 3.00 sec : ( +1.4037054845,  -4.7826081916)
... ...
At 4.00 sec : ( -2.7787118486,  -1.1997994809)
... ...
At 5.00 sec : ( -3.3781806892,  -0.8411792354)
... ...
```

⇖ wait, $\theta < -\pi$!?

SIMPLE ANIMATION

- It is easy to create **animations** with matplotlib. It is useful to demonstrate some of the results that suppose to “move” as a function of time!
- Here are a very simple example code to show how it works!

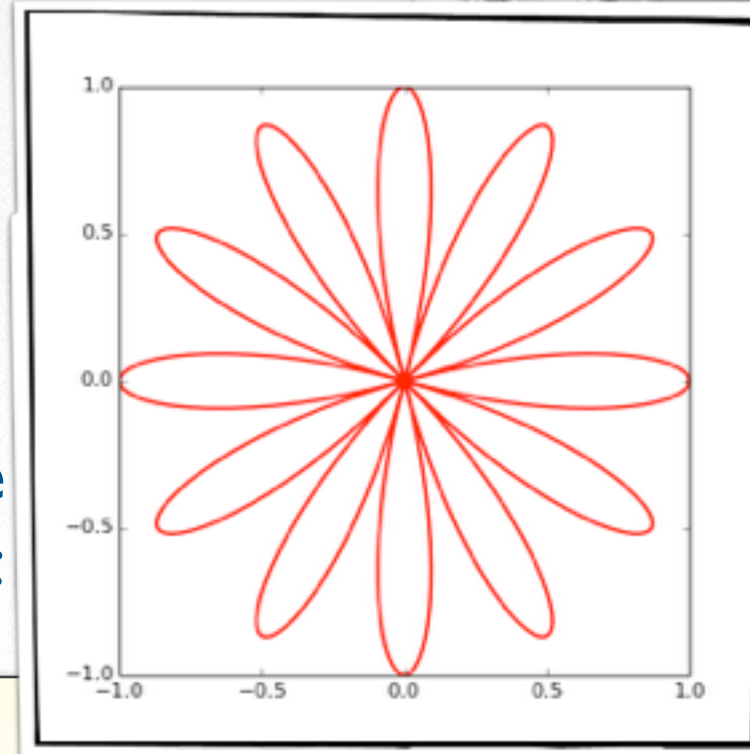
```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation   ⇐ import animation package

fig = plt.figure(figsize=(6,6), dpi=80)
ax = plt.axes(xlim=(-1.,+1.), ylim=(-1.,+1.))   ⇐ initial figure/axis

curve, = ax.plot([], [], lw=2, color='red')
                  ⇑ initial empty object(s)
```

partial l11-example-06.py

SIMPLE ANIMATION (II)

- The “core ” part of the code:

This is the output:




```
def init():
    curve.set_data([], [])
    return curve,

def animate(i):
    t = np.linspace(0.,np.pi*2.,400)
    x = np.cos(t*6.)*np.cos(t+2.*np.pi*i/360.)
    y = np.cos(t*6.)*np.sin(t+2.*np.pi*i/360.)

    curve.set_data(x, y)
    return curve,

anim = animation.FuncAnimation(fig, animate,
       init_func=init, frames=360, interval=40)
plt.show()
```

⇐ initial frame, all set to empty

⇐ have to return a tuple

⇖ update the data for frame index = **i**
(i is not an essential piece, it’s just a counter)

Initial an animation of total 360 frame ⇖⇑
with 40 mini-sec wait interval (=25 FPS)

partial l11-example-06.py

SOLVING ODE X ANIMATION

- "Merge" two previous codes as following:

```
fig = plt.figure(figsize=(6,6), dpi=80)
ax = plt.axes(xlim=(-1.2,+1.2), ylim=(-1.2,+1.2))

stick, = ax.plot([], [], lw=2, color='black')
ball,  = ax.plot([], [], 'ro', ms=10)
text   = ax.text(0.,1.1,'', fontsize = 16, color='black',
         ha='center', va='center')
```

⇖ initial empty objects:

```
m, g, R = 1., 9.8, 1.
t, h = 0., 0.001
y = np.array([np.pi*0.9999,0.])
```

⇐ Initial θ and θ'

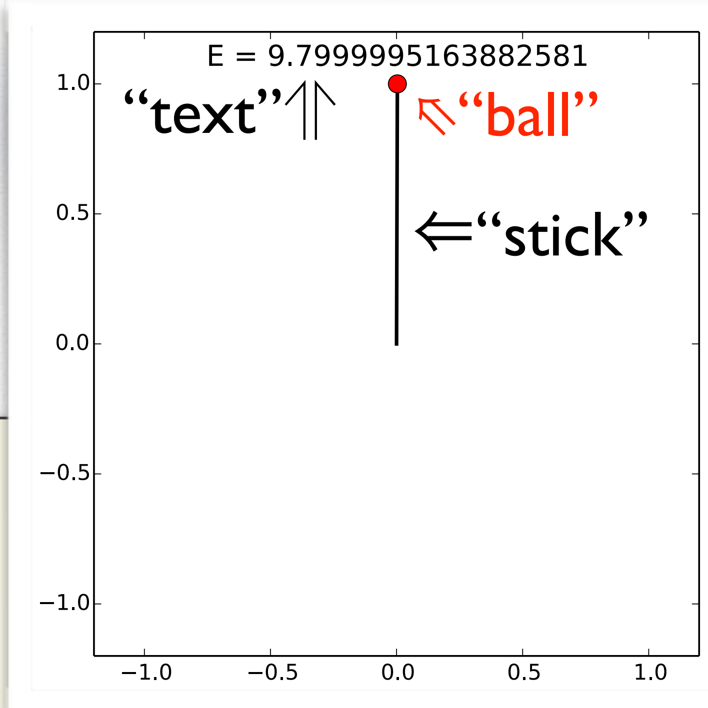
```
def f(t,y):
    theta   = y[0]
    thetap  = y[1]
    thetapp = -g/R*np.sin(theta)

    return np.array([thetap,thetapp])
```

⇐ function for calculating **θ'** and **θ''**

partial 111-example-05a.py

SOLVING ODE X ANIMATION (II)

- Core **animation + solving ODE**:

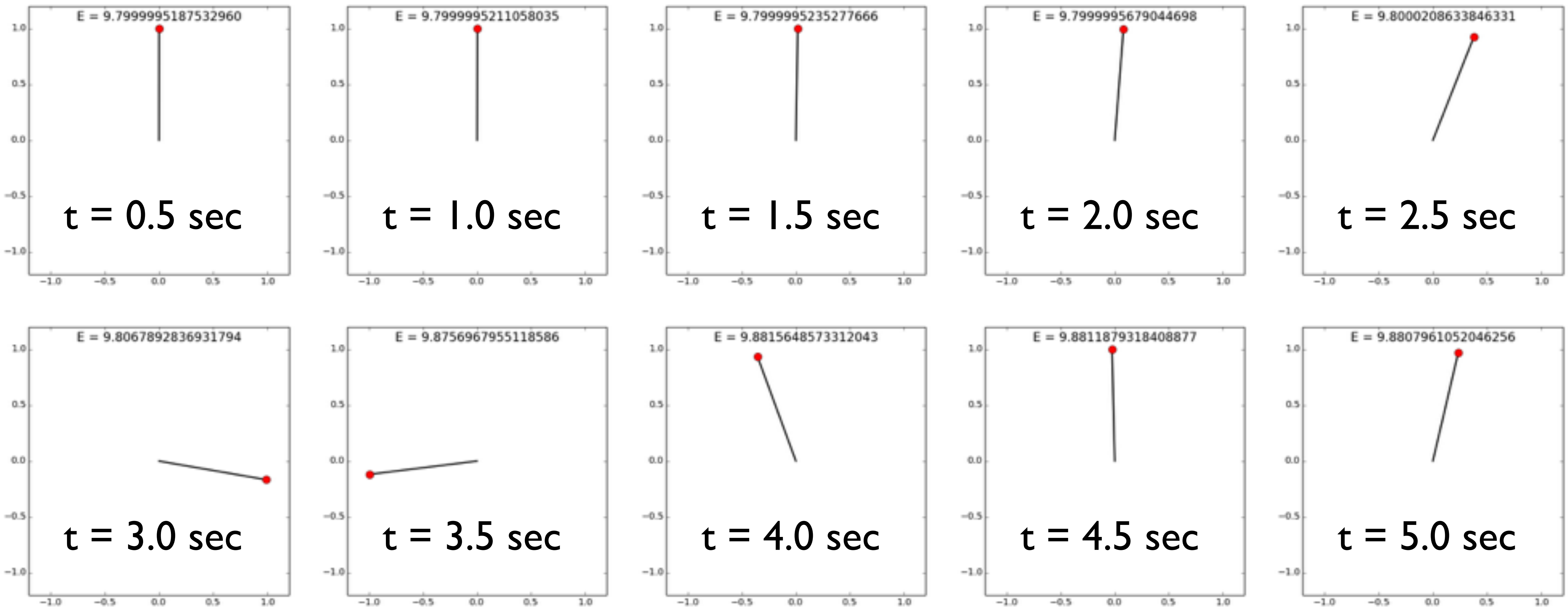



```
def animate(i):
    global t,y  ⇐ force t and y to be global variables
    for step in range(40):
        k1  = f(t, y)      ⇐ solve 40 steps
        y  += h*k1            (0.04 sec per frame)
        t  += h
    theta  = y[0]
    thetap = y[1]
    bx =  np.sin(theta)
    by = -np.cos(theta)
    ball.set_data(bx, by)             ⇐ plot the "ball" and "stick"
    stick.set_data([0.,bx], [0.,by])
    E = m*g*by + 0.5*m*(R*thetap)**2  ⇐ show the total energy
    text.set(text='E = %.16f' % E)
    return stick, ball, text
anim = animation.FuncAnimation(fig, animate, init_func=init,
       frames=10, interval=40)
```

partial 111-example-05a.py

DEMO TIME!

- **It moves!** But you will find the solver does not work too good almost immediately; the energy is not even conserved!



THAT'S WHY WE NEED A BETTER ODE SOLVER...

- One can simply replace the core part of the code to "upgrade" the ODE solutions.

RK2

```
for step in range(40):
    k1  = f(t, y)
    k2  = f(t+0.5*h, y+0.5*h*k1)
    y  += h*k2
    t  += h
```

partial 111-example-05b.py

RK4

```
for step in range(40):
    k1  = f(t, y)
    k2  = f(t+0.5*h, y+0.5*h*k1)
    k3  = f(t+0.5*h, y+0.5*h*k2)
    k4  = f(t+h, y+h*k3)
    y  += h/6.*(k1+2.*k2+2.*k3+k4)
    t  += h
```

partial 111-example-05c.py

This RK4 routine will not easily break the total energy cap easily at least.

USING THE ODE SOLVER FROM SCIPY

- The ODE solver under SciPy is also available in scipy.integrate module, together with the numerical integration tools:

SciPy.org Sponsored By ENTHOUGHT

Scipy.org | Docs | SciPy v0.14.0 Reference Guide

Integration and ODEs (scipy.integrate)

Integrators of ODE systems

odeint(func, y0, t[, args, Dfun, col_deriv, ...])	Integrate a system of ordinary differential equations.
ode(f[, jac])	A generic interface class to numeric integrators.
complex_ode(f[, jac])	A wrapper of ode for complex systems.

http://docs.scipy.org/doc/scipy/reference/integrate.html#module-scipy.integrate

USING THE ODE SOLVER FROM SCIPY (II)

```
import numpy as np
from scipy.integrate import ode⇐ import the module

m, g, R = 1., 9.8, 1.
t, y = 0., np.array([np.pi*0.9999,0.])⇐ now t and y are
                                         just initial conditions
def f(t,y):
    theta   = y[0]
    thetap  = y[1]   ⇐ exactly the same f(x,y)
    thetapp = -g/R*np.sin(theta)
    return np.array([thetap,thetapp])

intr = ode(f).set_integrator('dop853') ⇐ initialize the ode class
intr.set_initial_value(y, t)             with 'dop853' integrator

while intr.t<8.:
    intr.integrate(intr.t+0.1) ⇐ solve to current time + 0.1 sec

    theta  = intr.y[0]
    thetap = intr.y[1]
    print 'At %.2f sec : (%+14.10f, %+14.10f)' \
      % (intr.t, theta, thetap)
```

111-example-07.py

USING THE ODE SOLVER FROM SCIPY (III)

$\theta \downarrow$ $\dot{\theta} \downarrow$

```
At 0.10 sec : ( +3.1412629744,  -0.0003129294)
At 0.20 sec : ( +3.1412148812,  -0.0006567772)
At 0.30 sec : ( +3.1411294629,  -0.0010655165)
At 0.40 sec : ( +3.1409982801,  -0.0015795319)
At 0.50 sec : ( +3.1408083714,  -0.0022496097)
... ...
At 1.00 sec : ( +3.1379909749,  -0.0112320574)
... ...
At 1.50 sec : ( +3.1243942321,  -0.0538299284)
... ...
At 2.00 sec : ( +3.0593354818,  -0.2574312087)
... ...
At 2.50 sec : ( +2.7492944690,  -1.2202273084)
... ...
At 3.00 sec : ( +1.3819060253,  -4.8249634626)
... ...
At 4.00 sec : ( -2.7713127817,  -1.1525482114)
... ...
At 5.00 sec : ( -3.1253649922,  -0.0507902190)
... ...
```

- It's simply working smoothly.
- There are few more different integrator available in the scipy ode class, e.g. "vode", "zvode", etc.
- Please read the manual for details.

USING THE ODE SOLVER FROM SCIPY (IV)

- It's also pretty easy to merge the ODE solver with animation.

Initial the integrator

Replace the for-loop with a single commend

```
m, g, R = 1., 9.8, 1.
t = 0.
y = np.array([np.pi*0.9999,0.])

def f(t,y):
    theta   = y[0]
    thetap  = y[1]
    thetapp = -g/R*np.sin(theta)

    return np.array([thetap,thetapp])

intr = ode(f).set_integrator('dop853')
intr.set_initial_value(y, t)

def animate(i):
    intr.integrate(intr.t+0.040)

    theta  = intr.y[0]
    thetap = intr.y[1]
```

partial 111-example-07a.py

INTERMISSION

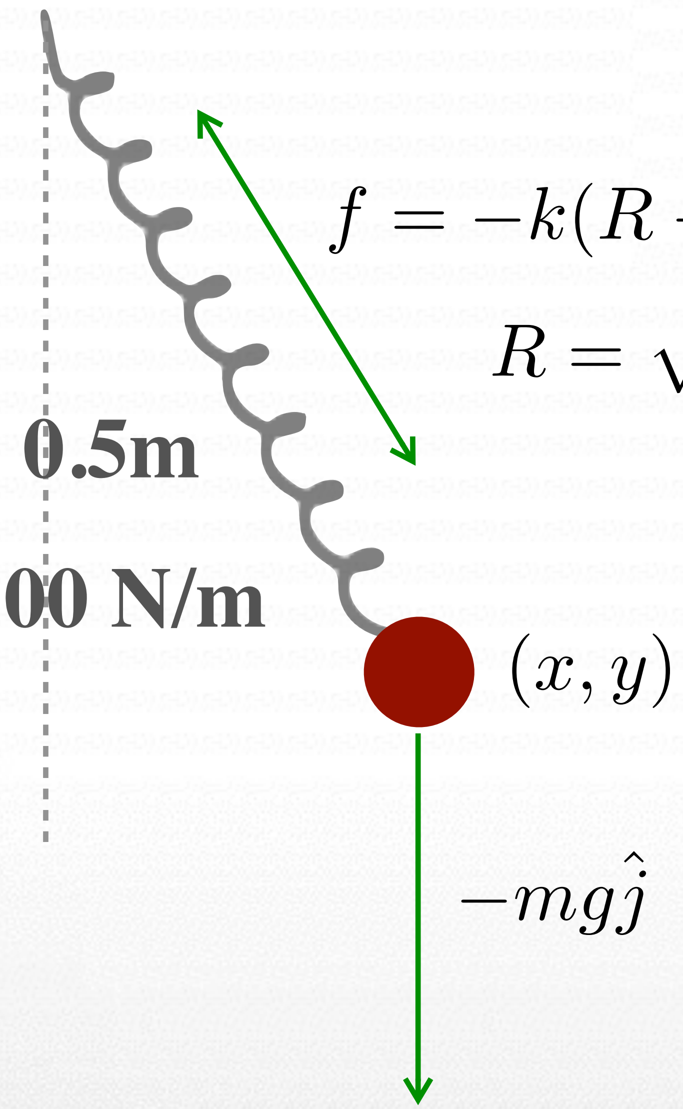
- What will happen if you given a critical initial condition to the preview simple pendulum example, e.g. $\theta = \pi$

$$\dot{\theta} = 0$$

- It could be fun if you can try to record the angle versus time (this can be done by a small modification to l11-example-07.py), and make a plot. If you set the initial condition to **a small angle** (when the small angle approximation still works), will you see if your solution close to a **sine/cosine function**?

FEW MORE EXAMPLES FOR YOUR AMUSEMENT

- Replace the “stick” with a spring:



$R_0 = 0.5\text{m}$

$k = 100\ \text{N/m}$

$m = 1\text{kg}$

$g = 9.8\text{m/s}^2$

$$f = -k(R - R_0)$$

$$R = \sqrt{x^2 + y^2}$$

$$f_x = f \cdot \frac{x}{R}\hat{i}$$

$$f_y = f \cdot \frac{y}{R}\hat{j}$$

Coordinate (x,y) is used instead of (R,θ) here.

Need to solve 4 equations (x,y,v_x,v_y) simultaneously

See
`l11-example-08.py`

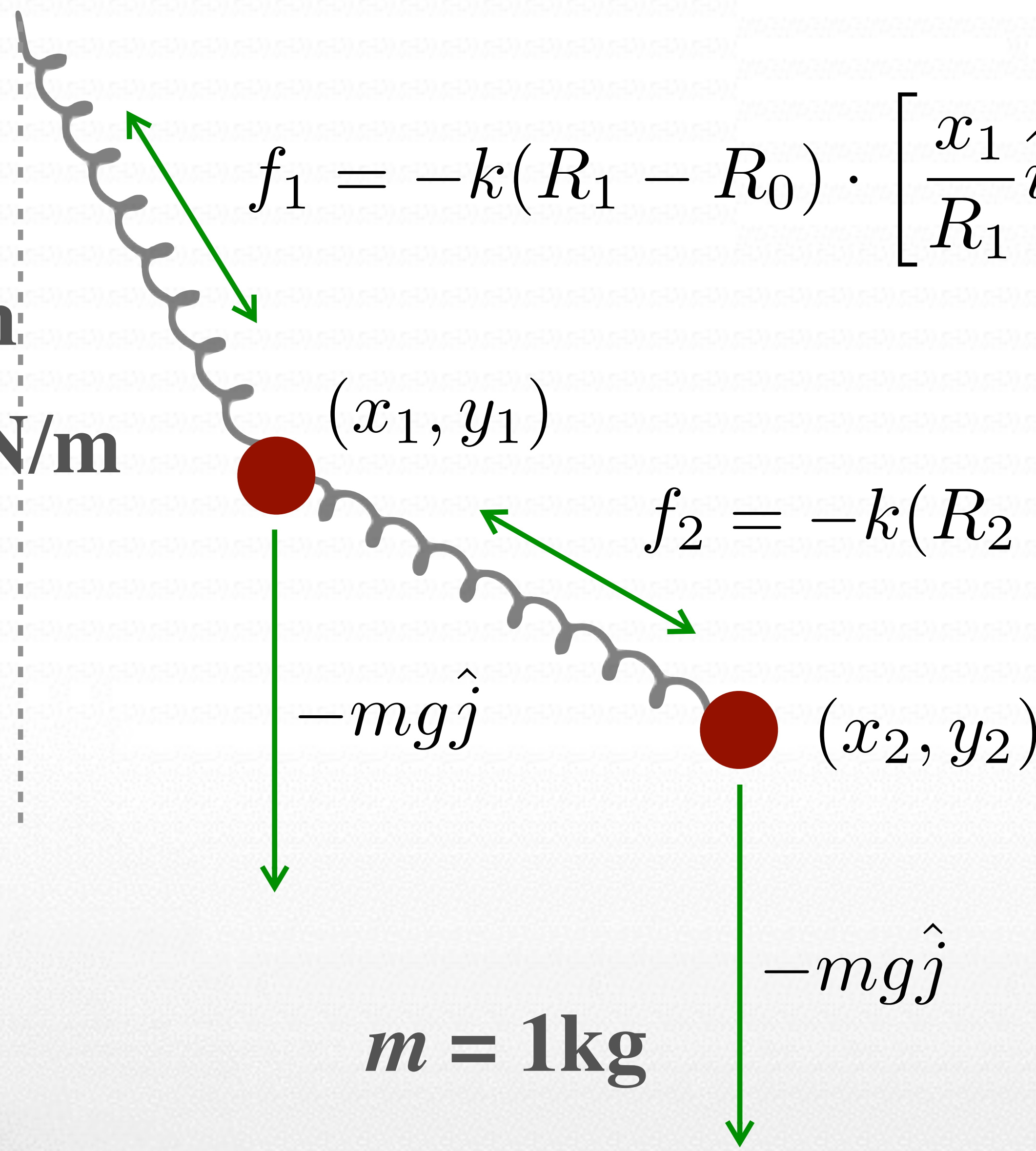
ONE IS COOL, TWO ARE CHAOTIC?

- A joint two-spring-ball system:

$$f_1 = -k(R_1 - R_0) \cdot \left[\frac{x_1}{R_1}\hat{i}, \frac{y_1}{R_1}\hat{j} \right]$$

$R_0 = 0.3$m

$k = 200$ N/m



$$f_2 = -k(R_2 - R_0) \cdot \left[\frac{(x_2 - x_1)}{R_2}\hat{i}, \frac{(y_2 - y_1)}{R_2}\hat{j} \right]$$

$m = 1$kg

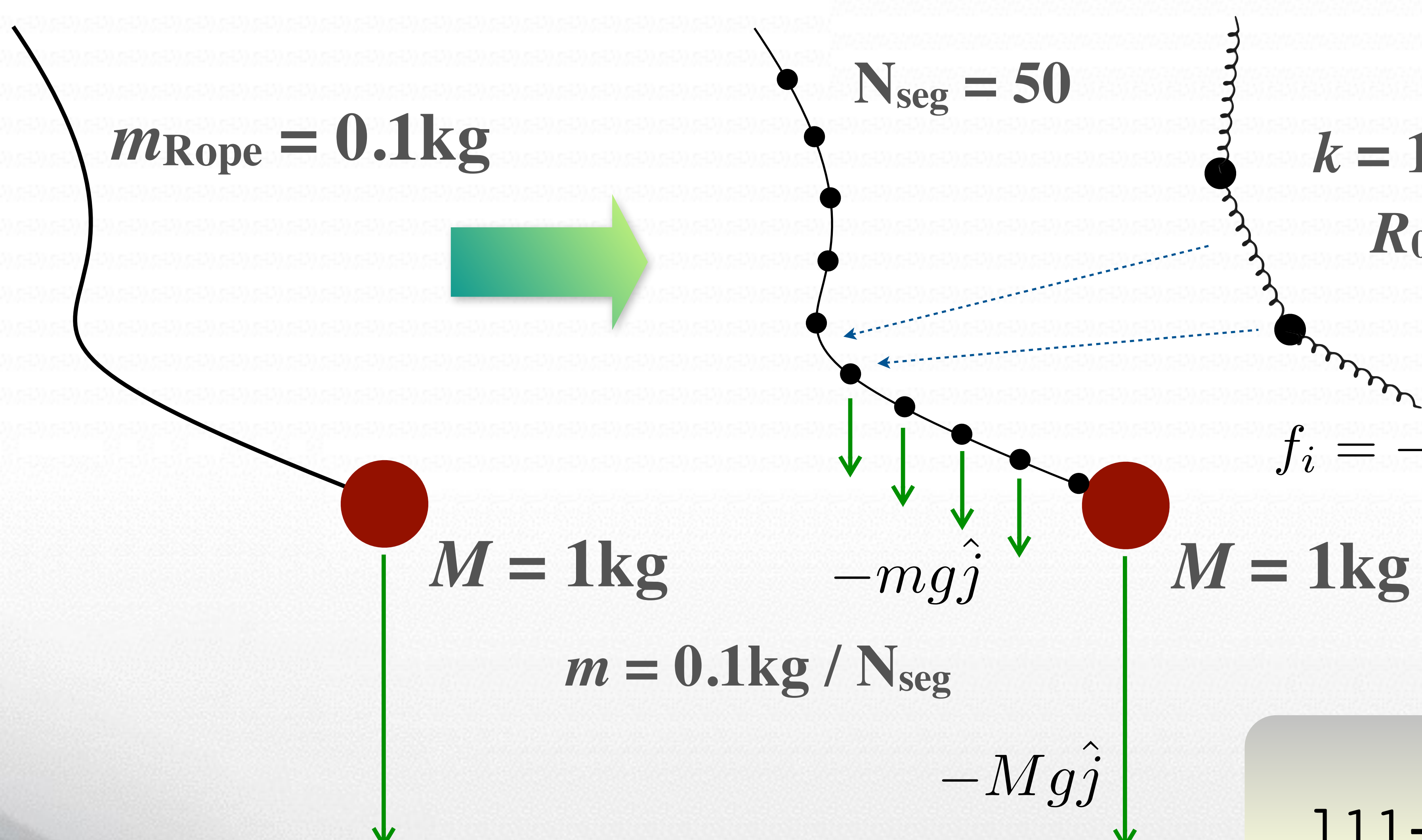
Need to solve 8 equations simultaneously

See `l11-example-08a.py`

A CHAIN OF SPRING-BALL = A ROPE?

- If we replace the “stick” with a rope, is it possible? Surely we need to use a simplified model to mimic a rope.

# of equations:
$N_{seg} * 4 = 200$



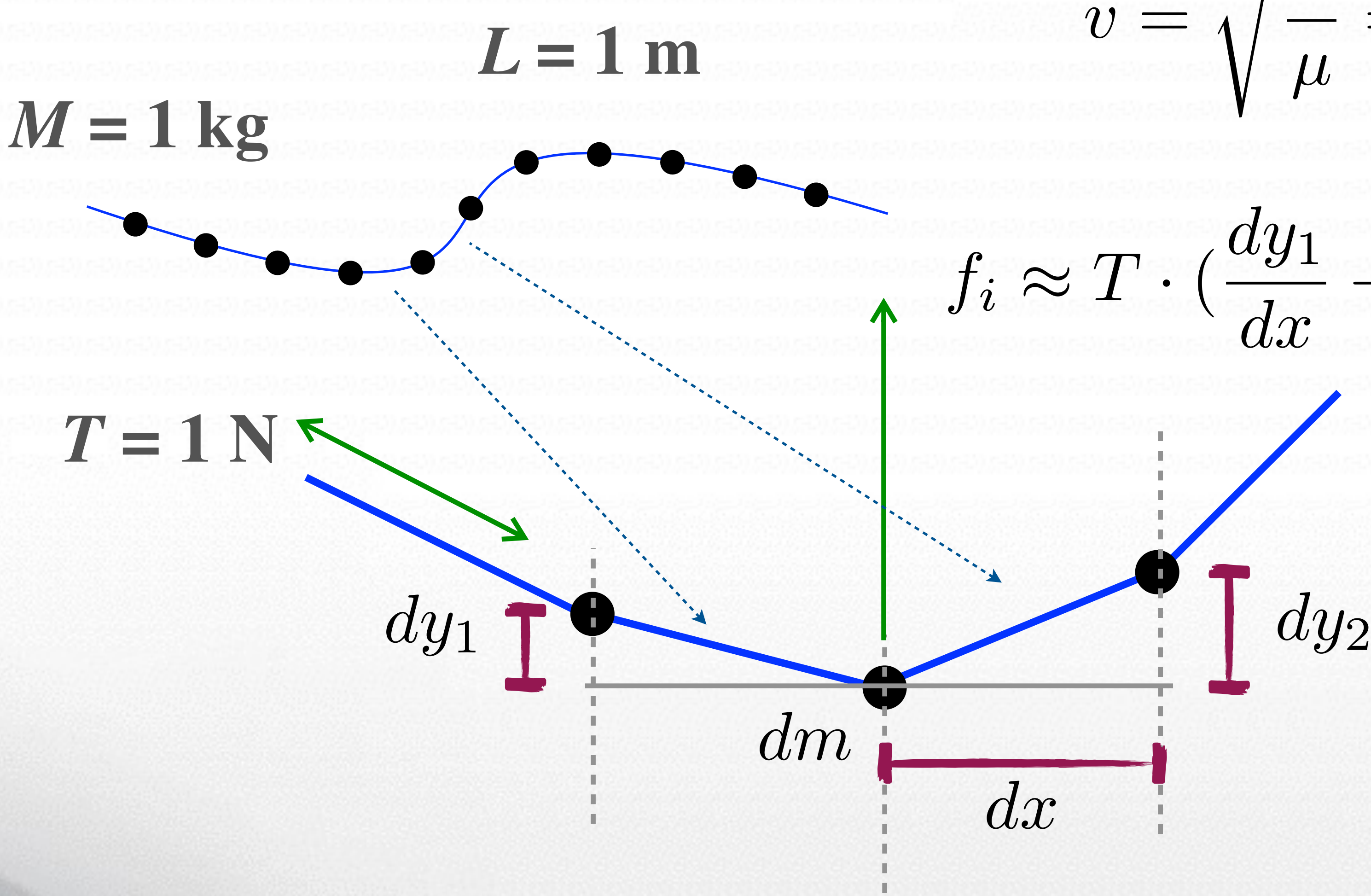
$$f_i = -k\Delta R_i \cdot \left(\frac{\Delta x_i}{R_i}\hat{i}, \frac{\Delta y_i}{R_i}\hat{j}\right)$$

See
`l11-example-08b.py`

WAVE ON A STRING

- Actually one can use a similar way to model a string — construct a N segment (massive) string and solve it with **small angle** approximation.

$$v = \sqrt{\frac{T}{\mu}} = \sqrt{\frac{TL}{M}} = f \cdot \lambda$$



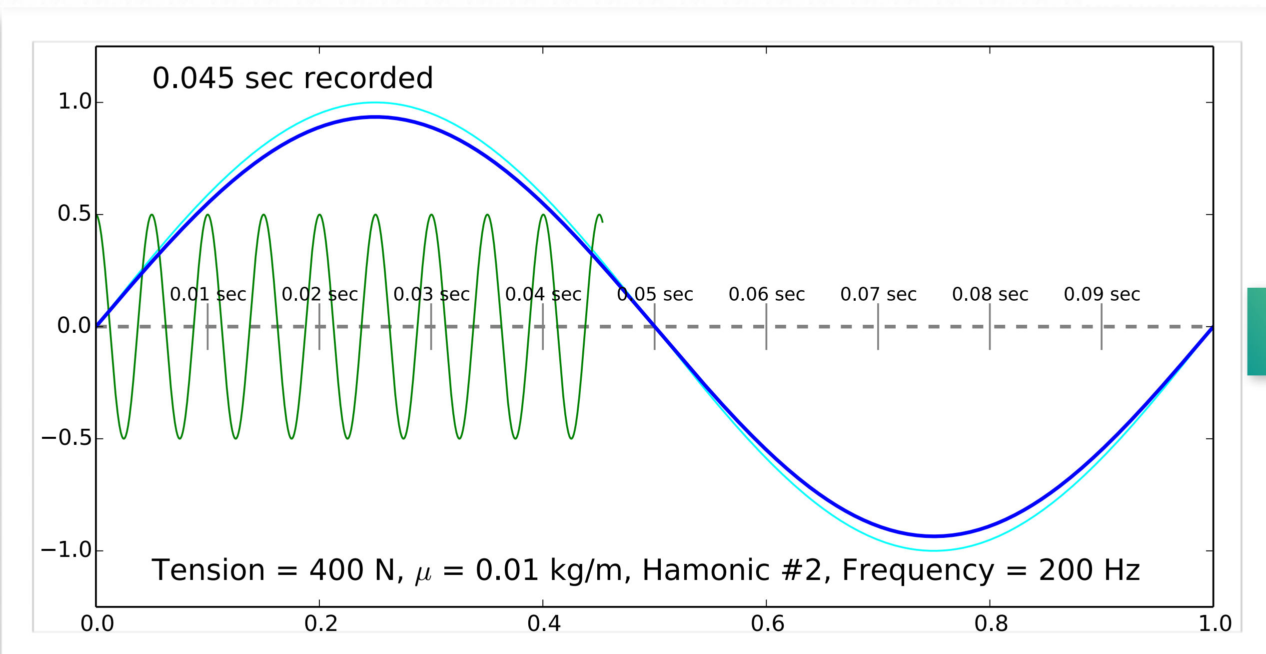
$$f_i \approx T \cdot (\frac{dy_1}{dx} + \frac{dy_2}{dx})$$

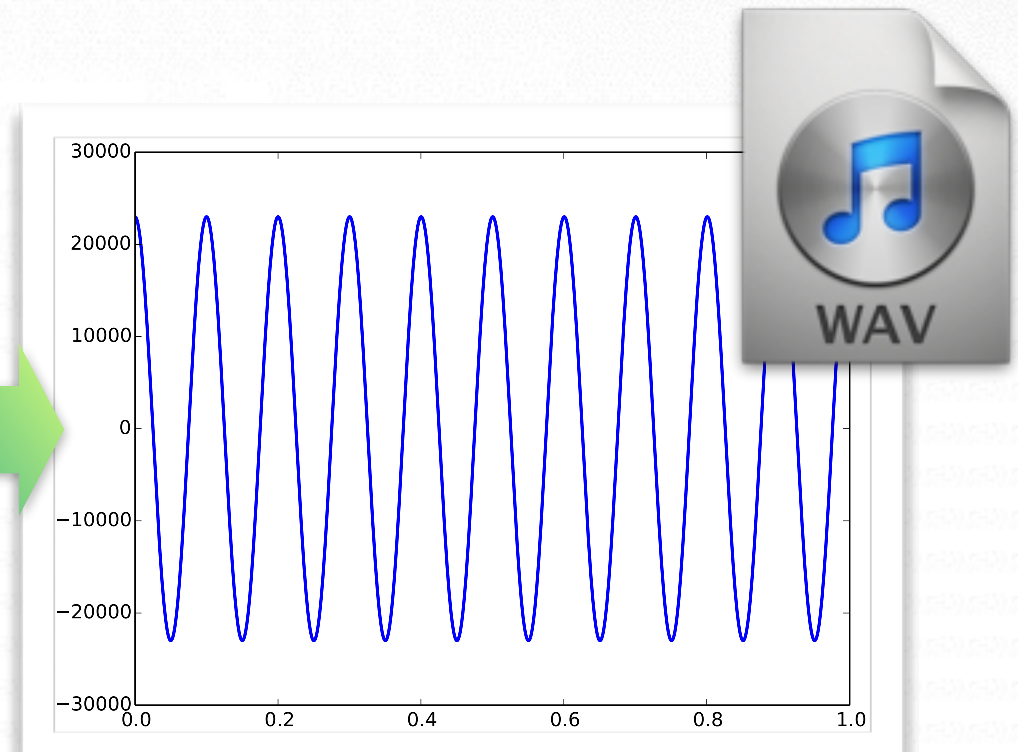
Set the initial condition to be simple sine waves and solve for the wave!

See `l11-example-09.py`

WAVE ON A STRING (II)

- It is also fun to record the vibration of the string, convert it to a wave file and play it out!





For the case of T = 400 N, μ= 0.01 kg/m, λ= 1m, we are expecting to hear a **200 Hz** sound!

See
`l11-example-09a.py`

COMMENTS

- We have demonstrated several interesting examples, surely you are encouraged to modify the code and test some different physics parameters, or different initial conditions.
- Basically all of those tasks can be easily done with the given ODE solver. In any case these are examples are VERY PHYSICS!
- Then – you may want to ask – how about PDEs? The general idea of PDE solving is similar but require some different implementations. There is no PDE solver available in SciPy yet. If you want, you can try the following packages:

  FiPy http://www.ctcms.nist.gov/fipy/
  SfePy http://sfepy.org/doc-devel/index.html

HANDS-ON SESSION

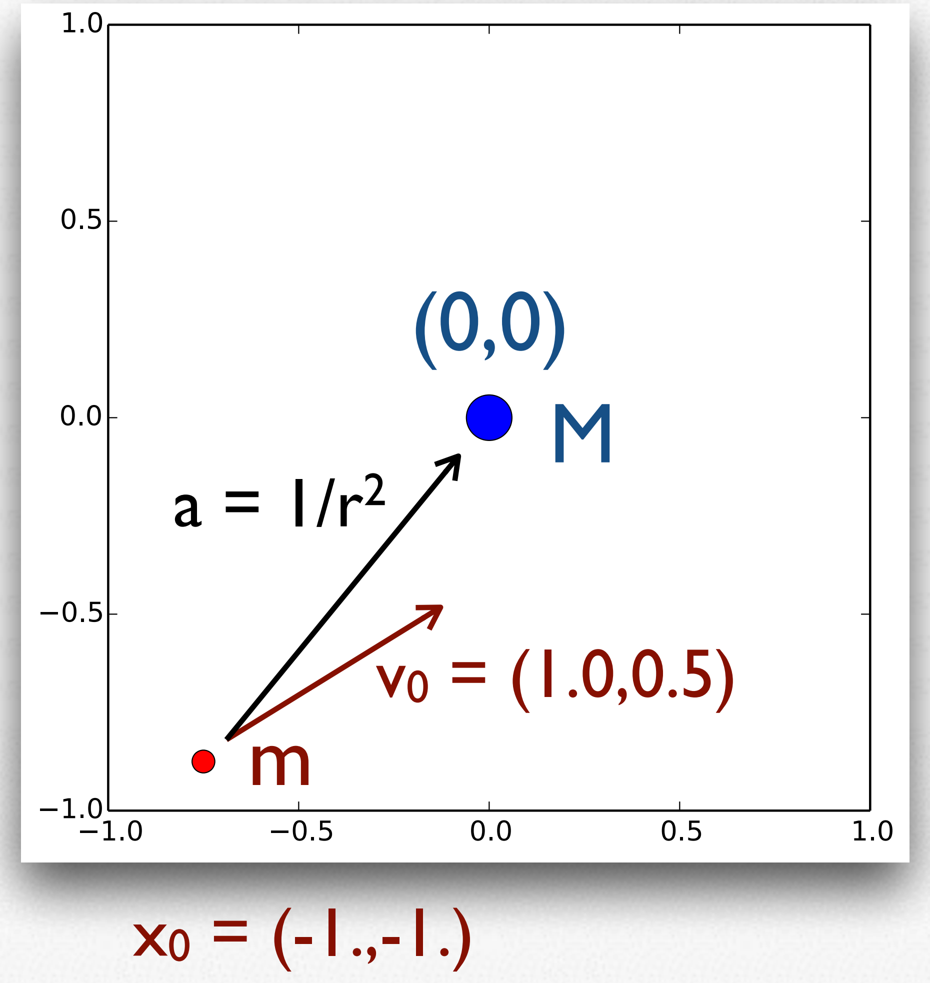
- **Practice 1:**
  Add some simple gravity to the system: there is a red star shooting toward the earth. Assuming the only acceleration between the earth and the red star is contributed by the gravitational force:

$$F = \frac{GMm}{r^2}$$

with G×M = 1. Thus:

$$a = \frac{dv}{dt} = F/m = \frac{1}{r^2}$$

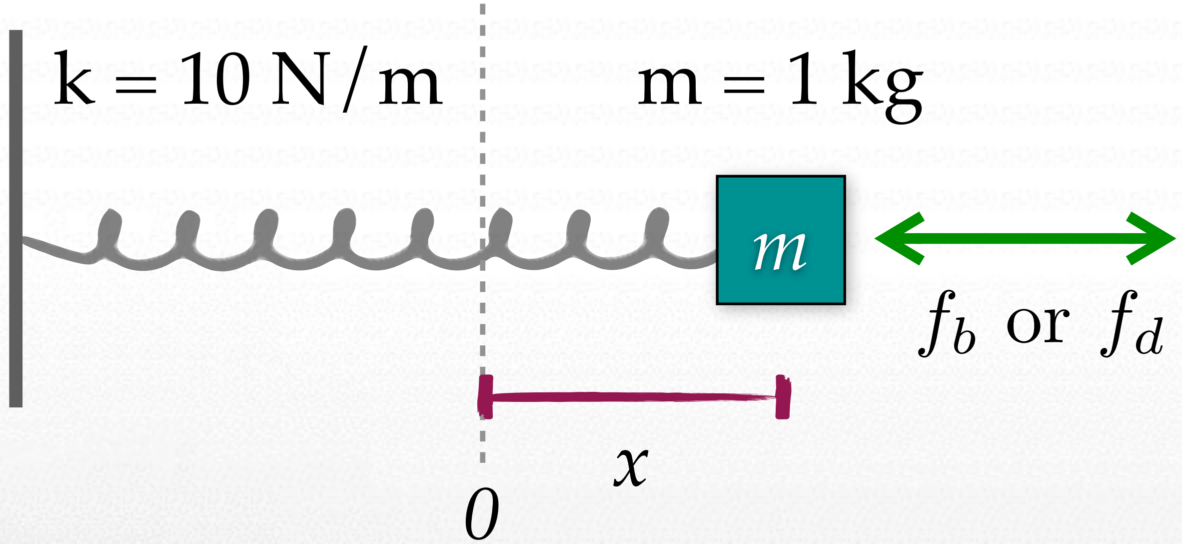
implement the code and produce the animation.



HANDS-ON SESSION

- **Practice 2:**
  damped or driven oscillators – please solve the following system with the extra (damping/driving) force and the given physics parameters.

Initial condition: **t = 0 sec, x = +0.1 m**



$$f_b = -b \cdot \frac{dx}{dt} \qquad b = 0.2 \text{ Ns/m}$$

or

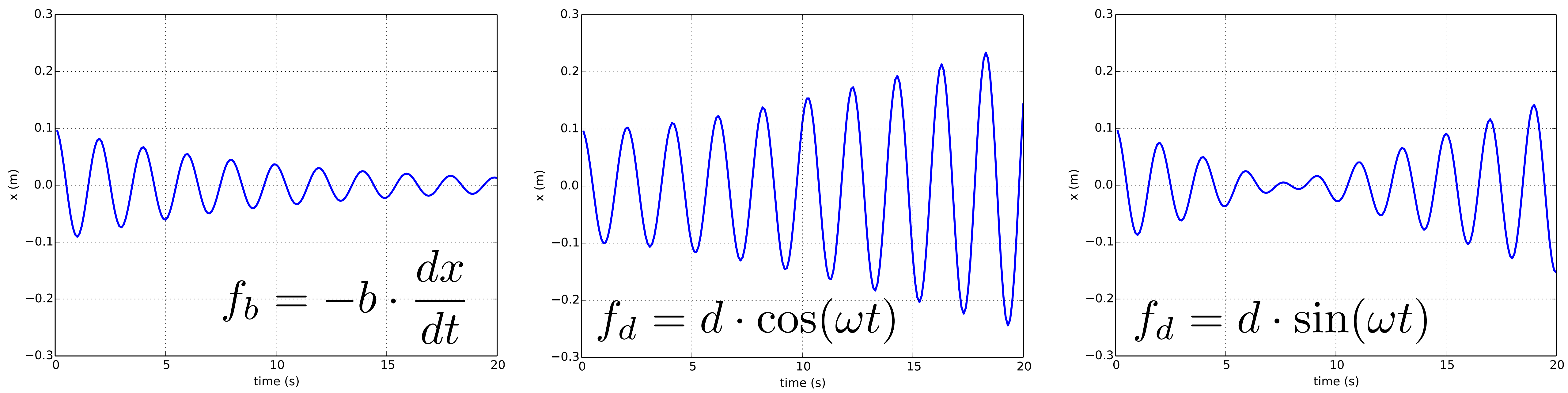
$$f_d = d \cdot \cos(\omega t) \qquad d = 0.08 \text{ N}$$

or

$$f_d = d \cdot \sin(\omega t) \qquad \omega = \pi \text{ rad/s}$$

HANDS-ON SESSION

- Please start with the given template on CEIBA. It can produce the following plots if you solve them correctly.



You may also play around with some what different physics parameters as well as the initial conditions.