2020

INTRODUCTION TO NUMERICAL ANALYSIS

Lecture 2-6: Solving ordinary differential equations

Kai-Feng Chen
National Taiwan University

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 — ***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**:

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

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

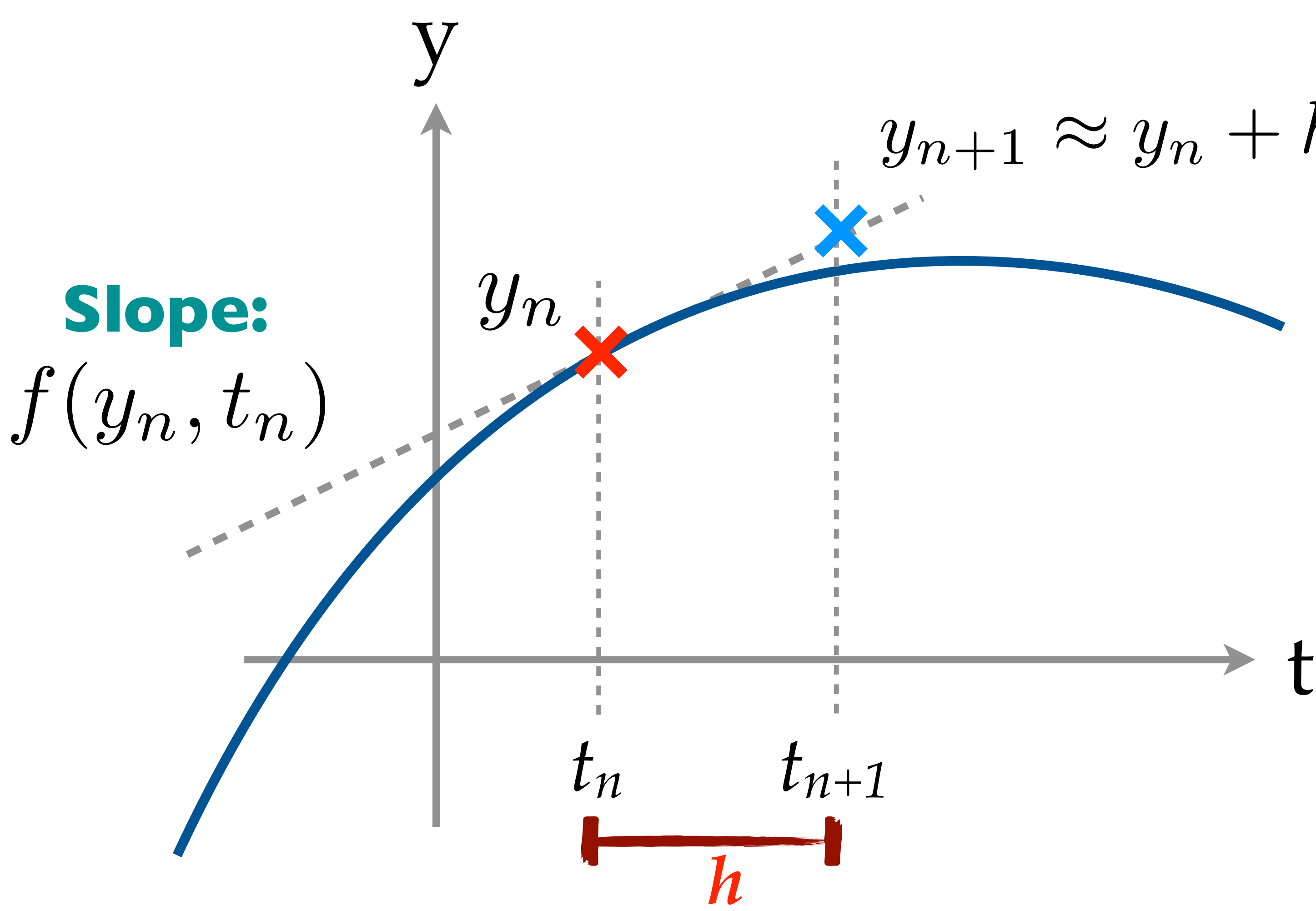
next step (y_{n+1}) **current step** (y_n)

For our trivial example: $$\frac{dy}{dt} = y \;\Rightarrow\; y_{n+1} \approx y_n + h \cdot y_n$$

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 $N \sim O(1/h)$ 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)))
```

l206-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)

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

It's just a number (slope)!

Insert the expansion into the integration:

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

Linear (first order) term must be cancelled

Insert the integral back:

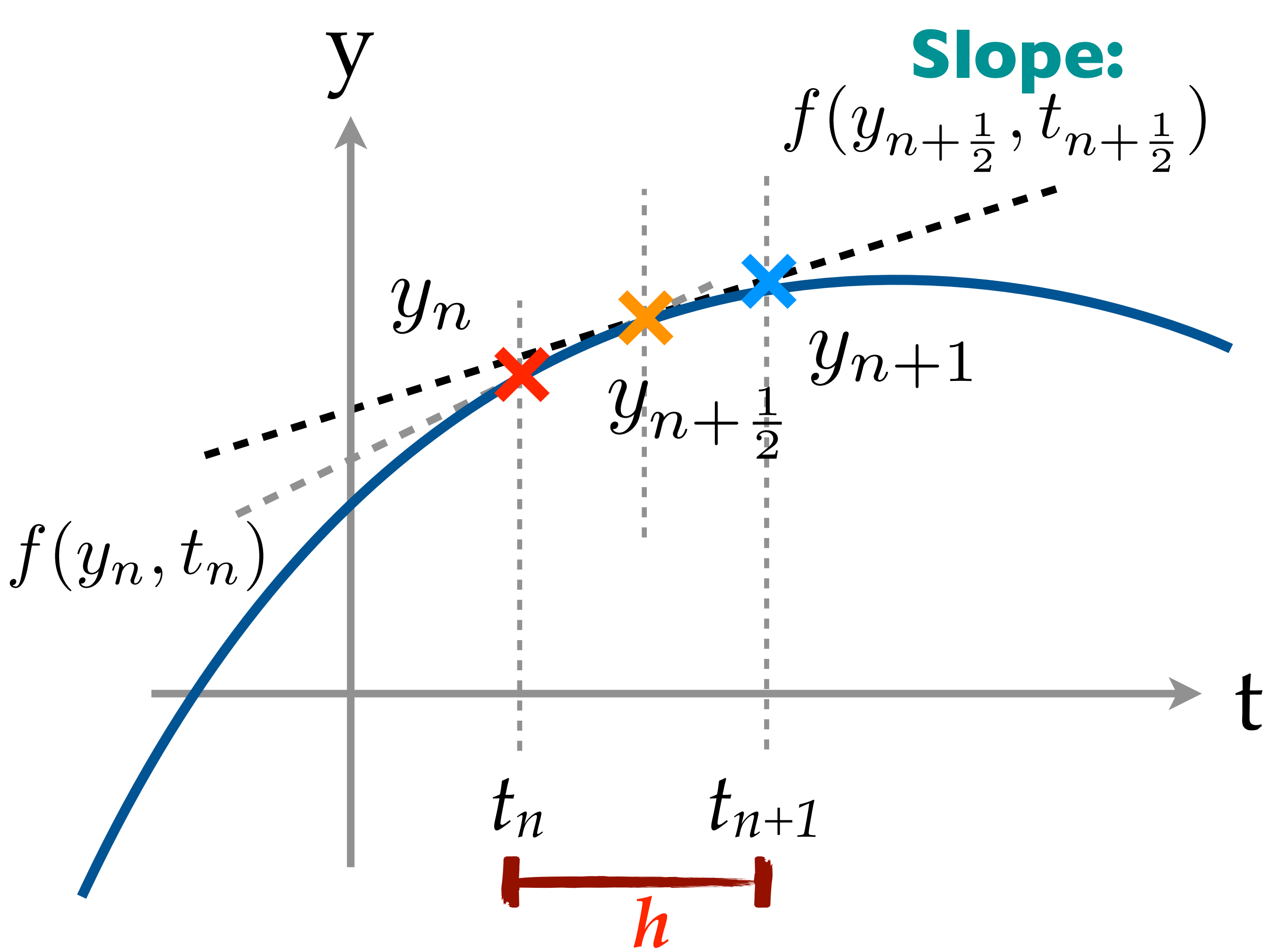
$$\int_{t_n}^{t_{n+1}} f(t,y)dt \approx h \cdot f(t_{n+\frac{1}{2}}, y_{n+\frac{1}{2}})$$

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

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:



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

Explicit formulae

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

IMPLEMENTATION OF "RK2"

- The coding is actually extremely simple:

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

l206-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.        ⇐ The same initial conditions & stepping
h = 0.001

while t<1.:
    k1  = f(t, y)                              RK4 solver
    k2  = f(t+0.5*h, y+0.5*h*k1)      ⇐ Simply calculate k1~k4 in a sequence
    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)     ⇐ Jump to the next step
    t  += h

y_exact = math.exp(t)
print('RK4 method: %.16f, exact: %.16f, diff: %.16f' % \
(y,y_exact,abs(y-y_exact)))
```

l206-example-03.py

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

⇐ Precision is of $O(h^4)$!

PRECISION EVOLUTION

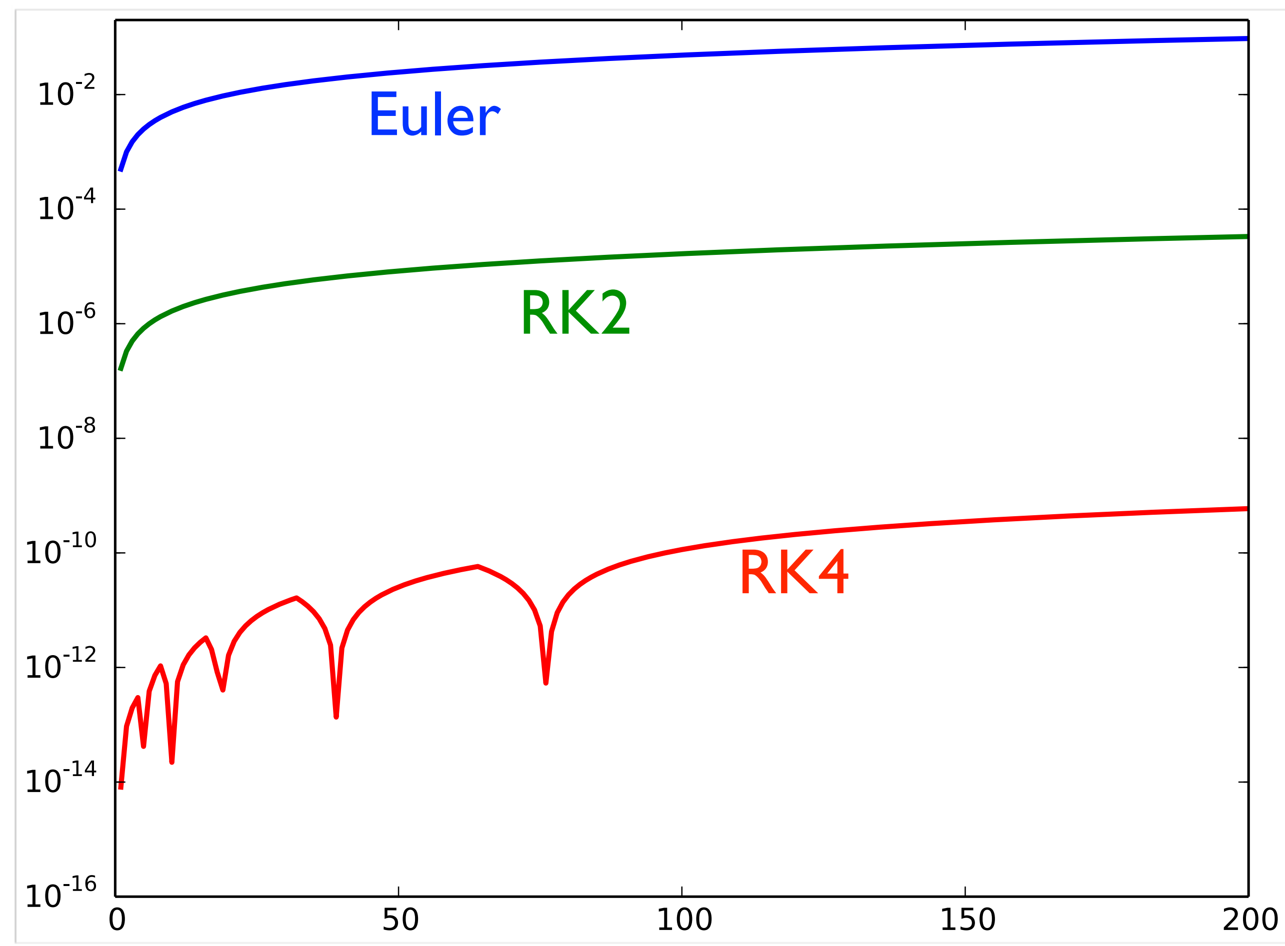
- Let's write a small code to demonstrate the "precision" of the solution as it evolves.
- You should be able to see the "accumulation" of numerical errors.

```
vt,y1,vy2,vy4 = [],[],[],[]   ⇐ List for storing the output
t = 0.
y1 = y2 = y4 = 1.
h = 0.001
while t<200.:   ⇐ now we calculate up to t=200

    k1  = f(t, y1)                               Euler method
    y1 += h*k1

    k1  = f(t, y2)                               RK2
    k2  = f(t+0.5*h, y2+0.5*h*k1)
    y2 += h*k2

    k1  = f(t, y4)                               RK4
    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.append(t)
    vy1.append(abs(y1-np.exp(t))/np.exp(t))
    vy2.append(abs(y2-np.exp(t))/np.exp(t))
    vy4.append(abs(y4-np.exp(t))/np.exp(t))

    ⇑ Store the relative errors
```

l206-example-03a.py (partial)

PRECISION EVOLUTION (II)

- Just make a simple plot.
- The initial uncertainties are of **O(h)**, **O(h²)**, and **O(h⁴)**.
- After 200,000 steps or more, the accumulated errors can be large.




```
plt.plot(vt,vy1,lw=2,c='Blue')
plt.plot(vt,vy2,lw=2,c='Green')
plt.plot(vt,vy4,lw=2,c='Red')
plt.yscale('log')
plt.ylim(1E-16,0.2)
plt.show()
```

⇑ Draw the relative differences

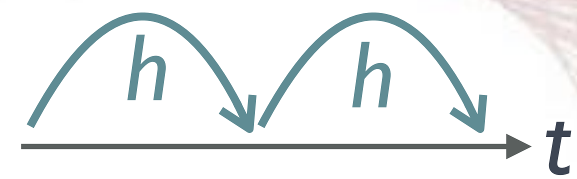
l206-example-03a.py (partial)

COMMENT: ADAPTIVE STEPPING

- If you check out the text books, they will tell you that "Although no one algorithm will work for all possible cases, **the fourth order Runge-Kutta method with adaptive step size** has proved to be robust and capable of industrial strength work."
- It is very similar to what people usually introduced in the numerical integration, by analyzing resulting errors and then adjust the step size in the routine.
- But how could we estimate the error in the ODE solving? In principle we could adopt a typical idea of "reduce the step size by a factor of two".

COMMENT: ADAPTIVE STEPPING (II)

- Look at the RK4 formulae:



Using the same step size h but move forward twice:

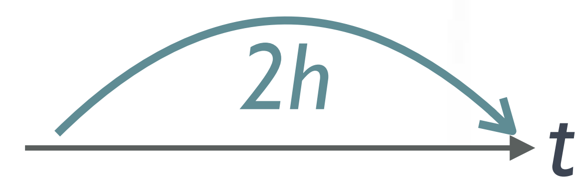
$$y_{n+1} \approx y_n + \ldots + (h)^5 \phi + O(h^6)$$

$$y_{n+2} \approx y_n + \ldots + 2(h)^5 \phi + O(h^6)$$

Using the step size 2h but move forward once:

$$y'_{n+2} \approx y_n + \ldots + (2h)^5 \phi + O(h^6)$$

- If we compare the two cases:



$$\Delta = y'_{n+2} - y_{n+2} = 30(h)^5 \phi + O(h^6)$$

- Although this idea works, but it's not really recommended/easy to carry it out directly. And when we estimate this error, we already **triple the steps**...

ADAPTIVE STEPPING: ERROR ESTIMATION

- Another way of error estimation: move to **5th order Runge-Kutta method, and compare the difference between 4th and 5th results**.

$$
\begin{aligned}
k_1 &= h \cdot f(t_n, y_n) \\
k_2 &= h \cdot f(t_n + a_2 h, y_n + b_{21} k_1) \\
&\ldots\ldots \\
k_6 &= h \cdot f(t_n + a_6 h, y_n + b_{61} k_1 + \ldots + b_{65} k_5)
\end{aligned}
$$

General RK5 formulae

$$y_{n+1}^{5^{\text{th}}\text{order}} = y_n + c_1 k_1 + c_2 k_2 + c_3 k_3 + c_4 k_4 + c_5 k_5 + c_6 k_6 + O(h^6)$$

$$y_{n+1}^{4^{\text{th}}\text{order}} = y_n + c_1^* k_1 + c_2^* k_2 + c_3^* k_3 + c_4^* k_4 + c_5^* k_5 + c_6^* k_6 + O(h^5)$$

compare these two equations

$$\Delta = y_{n+1}^{5^{\text{th}}\text{order}} - y_{n+1}^{4^{\text{th}}\text{order}}$$

ADAPTIVE STEPPING: COEFFICIENTS

- RK45 Coefficients (**Cash-Karp** version):

Cash-Karp Parameters for Embedded Runga-Kutta Method

i	a_i	b_{ij}					c_i	c_i^*
1							$\frac{37}{378}$	$\frac{2825}{27648}$
2	$\frac{1}{5}$	$\frac{1}{5}$					0	0
3	$\frac{3}{10}$	$\frac{3}{40}$	$\frac{9}{40}$				$\frac{250}{621}$	$\frac{18575}{48384}$
4	$\frac{3}{5}$	$\frac{3}{10}$	$-\frac{9}{10}$	$\frac{6}{5}$			$\frac{125}{594}$	$\frac{13525}{55296}$
5	1	$-\frac{11}{54}$	$\frac{5}{2}$	$-\frac{70}{27}$	$\frac{35}{27}$		0	$\frac{277}{14336}$
6	$\frac{7}{8}$	$\frac{1631}{55296}$	$\frac{175}{512}$	$\frac{575}{13824}$	$\frac{44275}{110592}$	$\frac{253}{4096}$	$\frac{512}{1771}$	$\frac{1}{4}$
$j =$		1	2	3	4	5		

IMPLEMENTATION OF "RK45"

- The code becomes somewhat "complex" now:

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

while t<1.:
  while(True):
    k1  = f(t, y)                                                    RK45 solver
    k2  = f(t+1./5.*h, y+1./5.*h*k1)
    k3  = f(t+3./10.*h, y+h*(3./40.*k1 + 9./40.*k2))
    k4  = f(t+3./5.*h, y+h*(3./10.*k1 - 9./10.*k2 + 6./5.*k3))
    k5  = f(t+h,y+h*(-11./54.*k1 + 5./2.*k2 - 70./27.*k3 +
          35./27.*k4))
    k6  = f(t+7./8.*h,y+h*(1631./55296.*k1 + 175./512.*k2 +
          575./13824.*k3 + 44275./110592.*k4 + 253./4096.*k5))

    yn = y+h*(37./378.*k1 + 250./621.*k3 + 125./594.*k4 +    ⇐ 5th order
              512./1771.*k6)
    yp = y+h*(2825./27648.*k1 + 18575./48384.*k3 +    ⇐ 4th order
              13525./55296.*k4 + 277./14336.*k5 + 1./4.*k6)
```

l206-example-04.py (partial)

IMPLEMENTATION OF "RK45" (CONT.)

- Then one has to scale the steps according to the **ERROR**:

```
        err = max(abs(yn-yp)/1E-14,0.01)
        if err<1.: break
```

Normalize the error to the desired precision; if accept, go for next step.

```
        hn = 0.9*h*err**-0.25
        if hn < h*0.1: hn = h*0.1
        h = hn
```

Shrinking the step size according to the error

```
    y = yn
    t += h
    steps += 1
```

enlarge the step size for the next iteration according to the error

```
    hn = 0.9*h*err**-0.2
    if hn > h*5.: hn = h*5.
    h = hn

y_exact = math.exp(t)
print('RK45 method after %d step (t=%.16f): %.16f, exact: %.16f, diff: %.16f' % (steps,t,y,y_exact,abs(y-y_exact)))
```

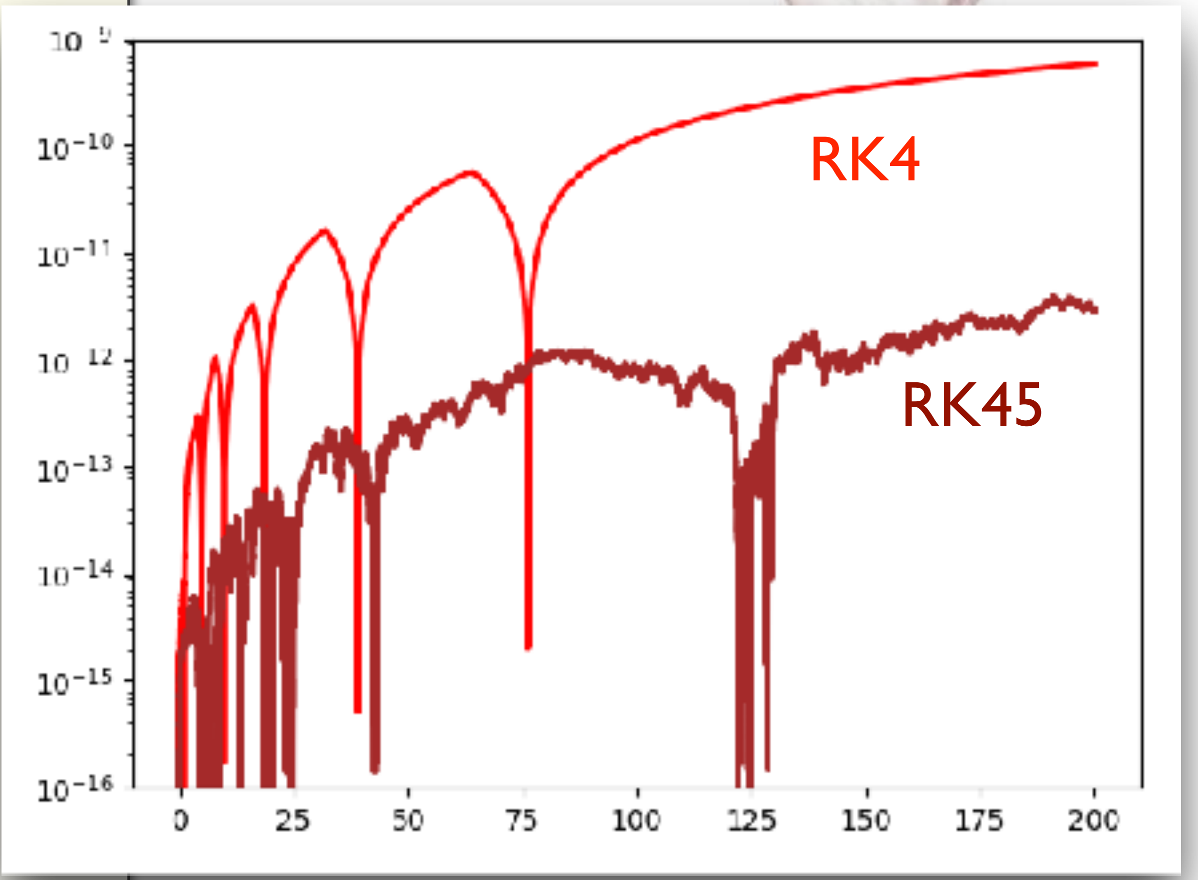
```
RK45 method after 147 step
    (t=1.0052500244037599):
        2.7325904017088232,
exact:  2.7325904017088298,
diff:   0.0000000000000067
```

l206-example-04.py (partial)

PRECISION EVOLUTION (AGAIN)

```
vt4,vy4 = [],[]
vt45,vy45 = [],[]

t, y = 0., 1.
h = 0.001
while t<200.:                                    RK4
. . . . . .
  vt4.append(t)
  vy4.append(abs(y-np.exp(t))/np.exp(t))

t, y = 0., 1.
h = 0.001
while t<200.:                                    RK45
. . . . . .
  vt45.append(t)
  vy45.append(abs(y-np.exp(t))/np.exp(t))

plt.plot(vt4,vy4,lw=2,c='Red')
plt.plot(vt45,vy45,lw=2,c='Brown')
plt.yscale('log')
plt.ylim(1E-16,1E-9)
plt.show()
```

l206-example-04a.py (partial)



- One can see the “RK45” method w/ adaptive steps further improves the precision!

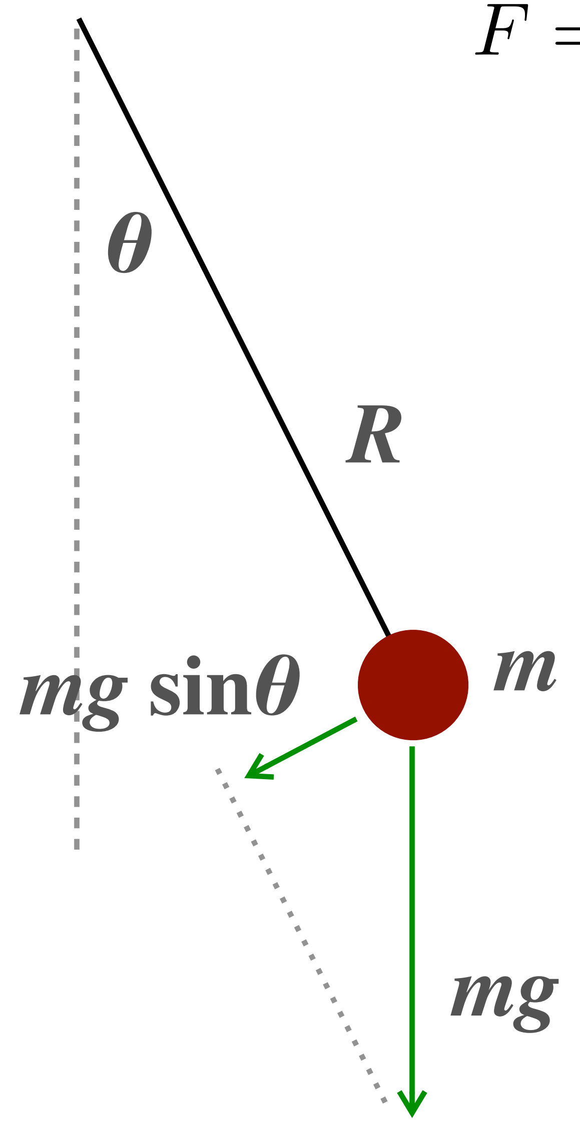
INTERMISSION

- 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 (l206-example-03a.py or l206-example-04a.py) and see how the error accumulated along with steps for a different differential equation.

A LITTLE BIT OF PHYSICS: SIMPLE PENDULUM



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

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

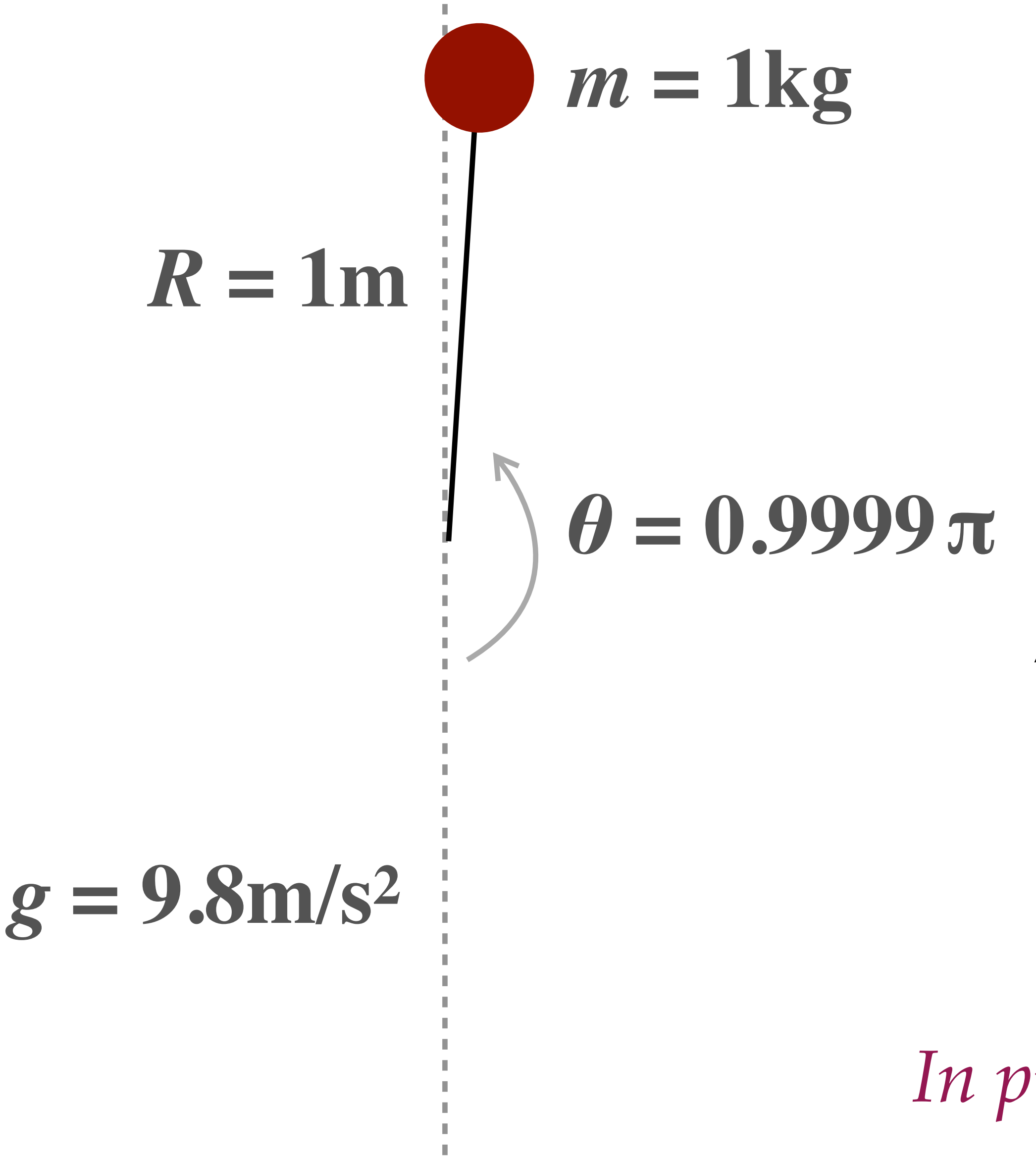
Solving 2nd order ODE =

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))
```

l206-example-05.py

SOLVE FOR 2 ODE'S TOGETHER (II)

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

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

```
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, θ<–π!?

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

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)
```

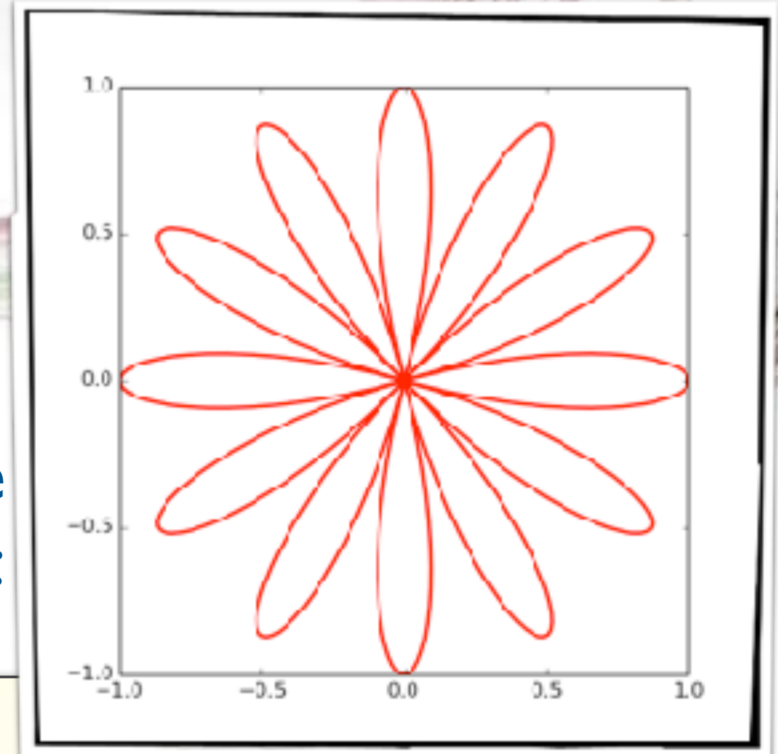
l206-example-06.py (partial)

You can also use **vpython** to create the animations!
(I know some of you already learned it before!)

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)

l206-example-06.py (partial)

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'

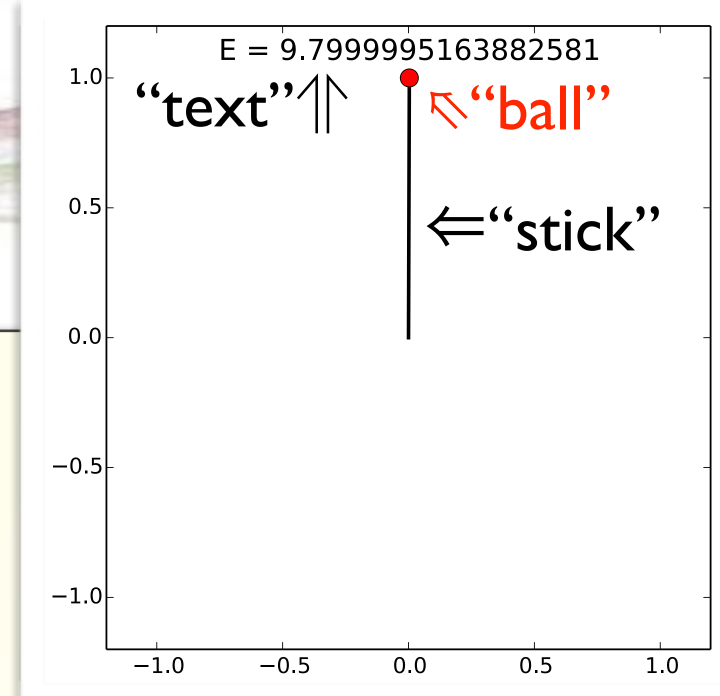
```
def f(t,y):
    theta   = y[0]
    thetap  = y[1]
    thetapp = -g/R*np.sin(theta)

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

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

l206-example-07.py (partial)

SOLVING ODE X ANIMATION (II)

- Core **animation + solving ODE**:

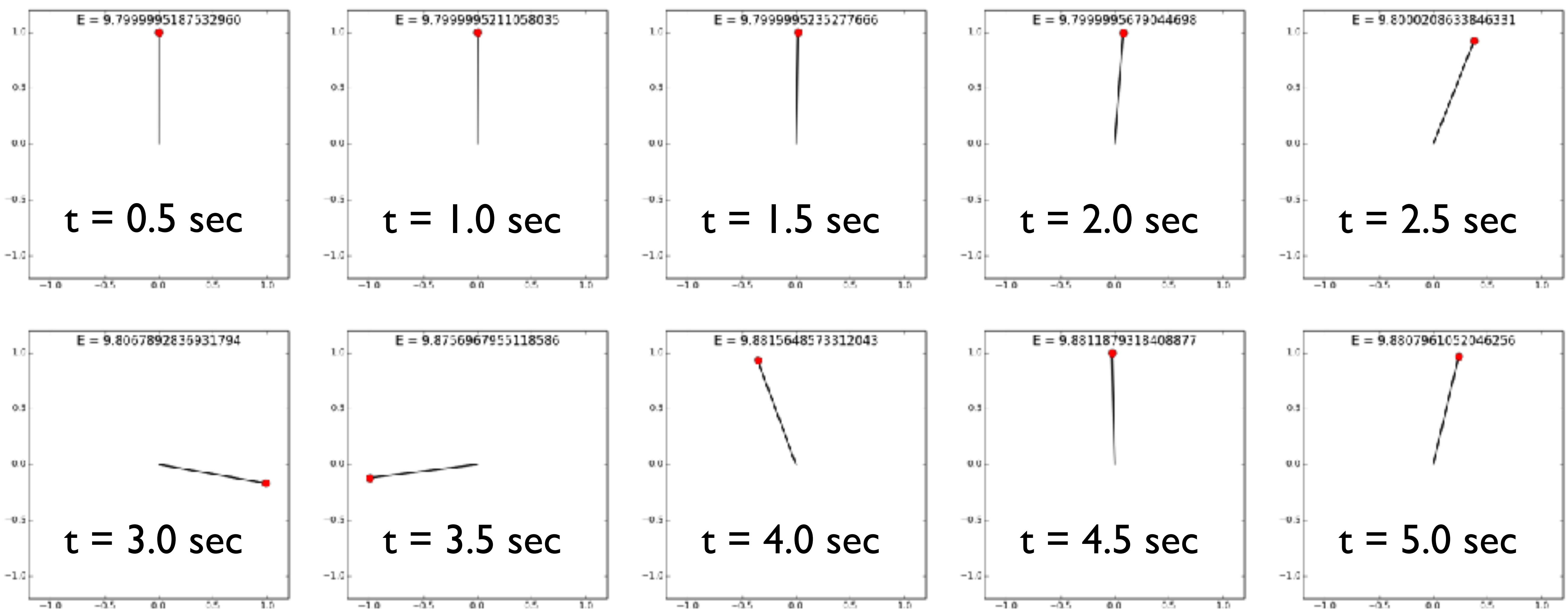



```
def animate(i):
    global t,y          ⇐ force t and y to be global variables

    for step in range(40):⇐ solve 40 steps
        k1  = f(t, y)       (0.04 sec per frame)
        y   += h*k1
        t   += h

    theta  = y[0]
    thetap = y[1]

    bx =  np.sin(theta)
    by = -np.cos(theta)
    ball.set_data(bx, by)
    stick.set_data([0.,bx], [0.,by]) ⇐ plot the "ball" and "stick"

    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)
```

l206-example-07.py (partial)

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.

```
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
```

l206-example-07a.py (partial)

RK2

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
```

l206-example-07b.py (partial)

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 v1.0.0 Reference Guide

Integration and ODEs (scipy.integrate)

Solving initial value problems for ODE systems

The solvers are implemented as individual classes which can be used directly (low-level usage) or through a convenience function.

solve_ivp(fun, t_span, y0[, method, t_eval, ...])	Solve an initial value problem for a system of ODEs.
RK23(fun, t0, y0, t_bound[, max_step, rtol, ...])	Explicit Runge-Kutta method of order 3(2).
RK45(fun, t0, y0, t_bound[, max_step, rtol, ...])	Explicit Runge-Kutta method of order 5(4).

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 solve_ivp   ⇐ import the routine

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

    return np.array([thetap,thetapp])

while t<8.:
    sol = solve_ivp(f, [t, t+0.1], y)   ⇐ solve to current time + 0.1 sec

    y      = sol.y[:,-1]
    t      = sol.t[-1]
    theta  = y[0]
    thetap = y[1]

    print('At %.2f sec : (%+14.10f, %+14.10f)' % (t, theta, thetap))
```

l206-example-08.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 working smoothly!
- The default algorithm is **RK45** (*as we introduced earlier, the error is controlled assuming 4th order accuracy, but steps are taken using a 5th formula*).
- Few other different methods are also available.

USING THE ODE SOLVER FROM SCIPY (IV)

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

Replace the for-loop with a single commend

call the integrator

```
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])

def animate(i):
    global t, y

    sol = solve_ivp(f, [t, t+0.040], y)
    y   = sol.y[:,-1]
    t   = sol.t[-1]

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

l206-example-08a.py (partial)

ANIMATION WITH VPYTHON

- **VPython** is an easy tool to create 3D displays and animations.
- I believe some of you are quite familiar with it already! So here we will just introduce it briefly and connect it with scipy ODE solver as a demonstration.
- Installation of VPython:
  - In your terminal run this command, which will install VPython 7 for your python environment:

```
> pip install vpython
```

  - Or if you are using Anaconda:

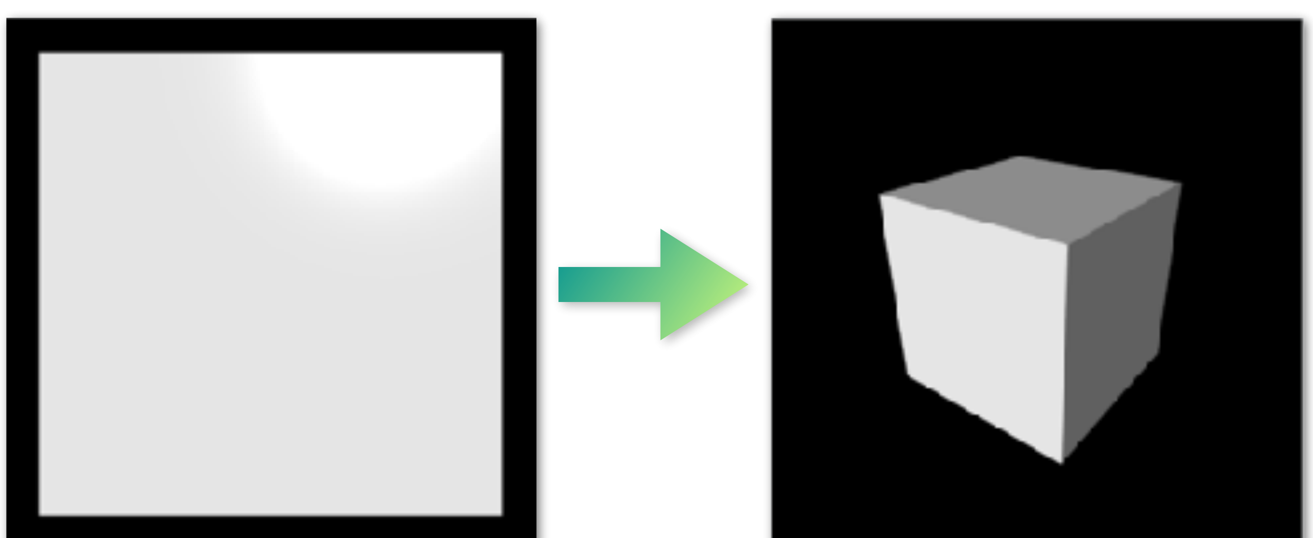
```
> conda install -c vpython vpython
```

ANIMATION WITH VPYTHON (II)

- A minimal VPython example:

```
>>> from vpython import *
>>> scene = canvas(width=480, height=480)
>>> cube = box(pos=vector(0.,0.,0.))
```

and this should give you a cube shown in your browser. (Remark: in old version of VPython it should show in a window!)



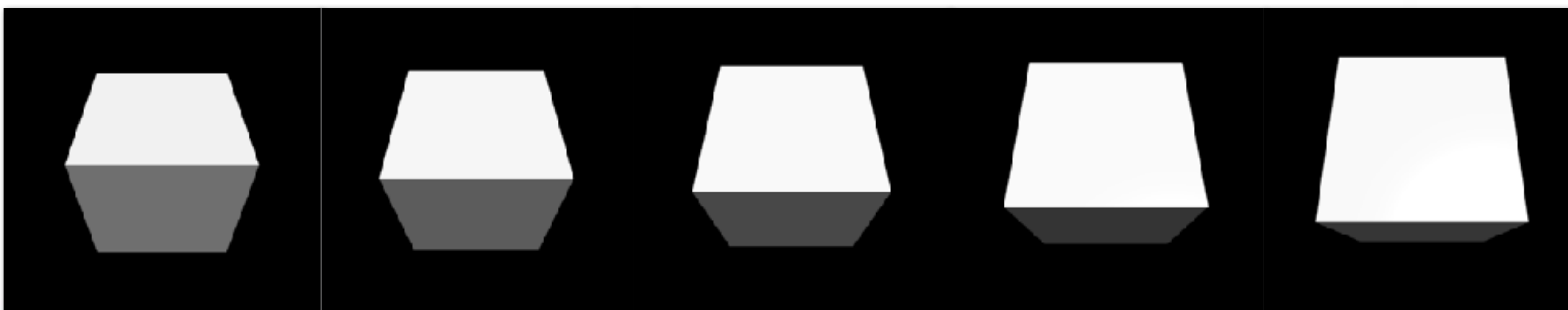
Zoom & rotate
the scene a little bit!

ANIMATION WITH VPYTHON (III)

- Now we shall make it **animated**!

```
>>> while True:
...       cube.rotate(angle=0.01)
...       rate(25.) ⇐ frequency = 25 : halt the computation for 1/25 sec
```

and this will give you a rotating cube, shown in your browser!



Now we can integrate VPython with our ODE solutions and make a proper animation!

VPYTHON + SCIPY

```
import numpy as np
from vpython import *
from scipy.integrate import solve_ivp

scene = canvas(width=480, height=480)
floor = box(pos=vector(0.,-1.1,0.), length=2.2, height=0.01,
width=1.2, opacity=0.2)
ball = sphere(radius=0.05, color=color.red)
rod = cylinder(pos=vector(0.,0.,0.),axis=vector(1,0,0),
radius=0.01, color=color.white)
txt = label( pos=vec(0,1.4,0), text='', line=False)

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

def f(t,y):
. . . . . .
    return np.array([thetap,thetapp])
```

a "floor" box for showing the ground in the scene

⇑ all the VPython objects

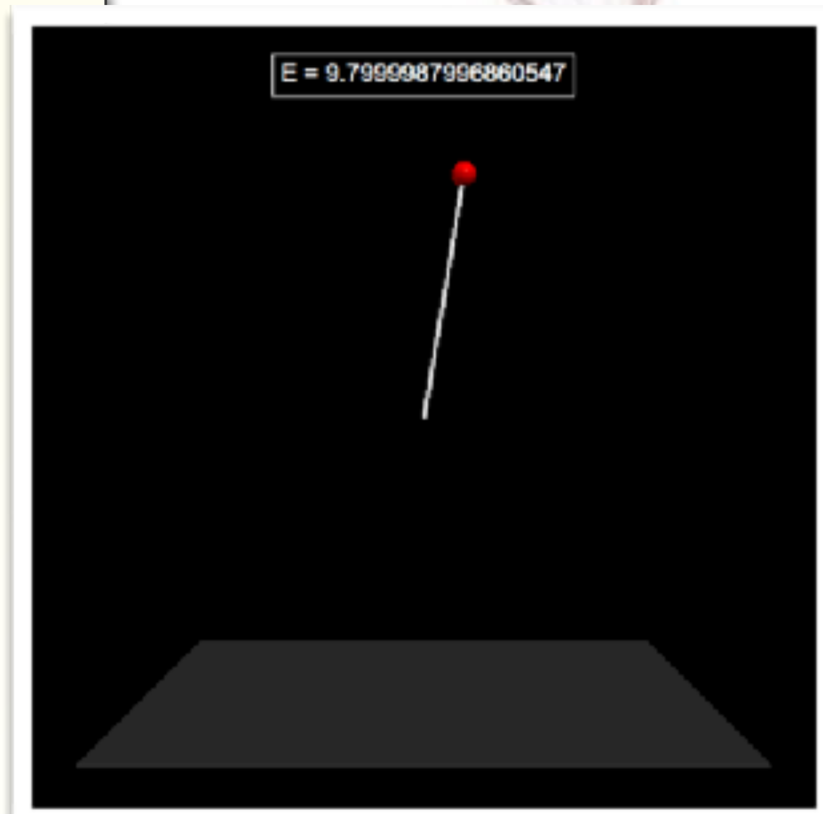
l206-example-08b.py (partial)

VPYTHON + SCIPY (II)

- The main ODE solving + animation loop — simply calculate the resulting theta and convert it to the coordination.

```
while True:
    sol = solve_ivp(f, [t, t+0.040], y)
    y   = sol.y[:,-1]
    t   = sol.t[-1]

    theta    = y[0]
    thetap   = y[1]

    ball.pos.x =  np.sin(theta)
    ball.pos.y = -np.cos(theta)
    rod.axis = ball.pos

    E = m*g*ball.pos.y + 0.5*m*(R*thetap)**2
    txt.text = 'E = %.16f' % E

    rate(1./0.040)
```

⇑ call the ODE solver



l206-example-08b.py (partial)

INTERMISSION

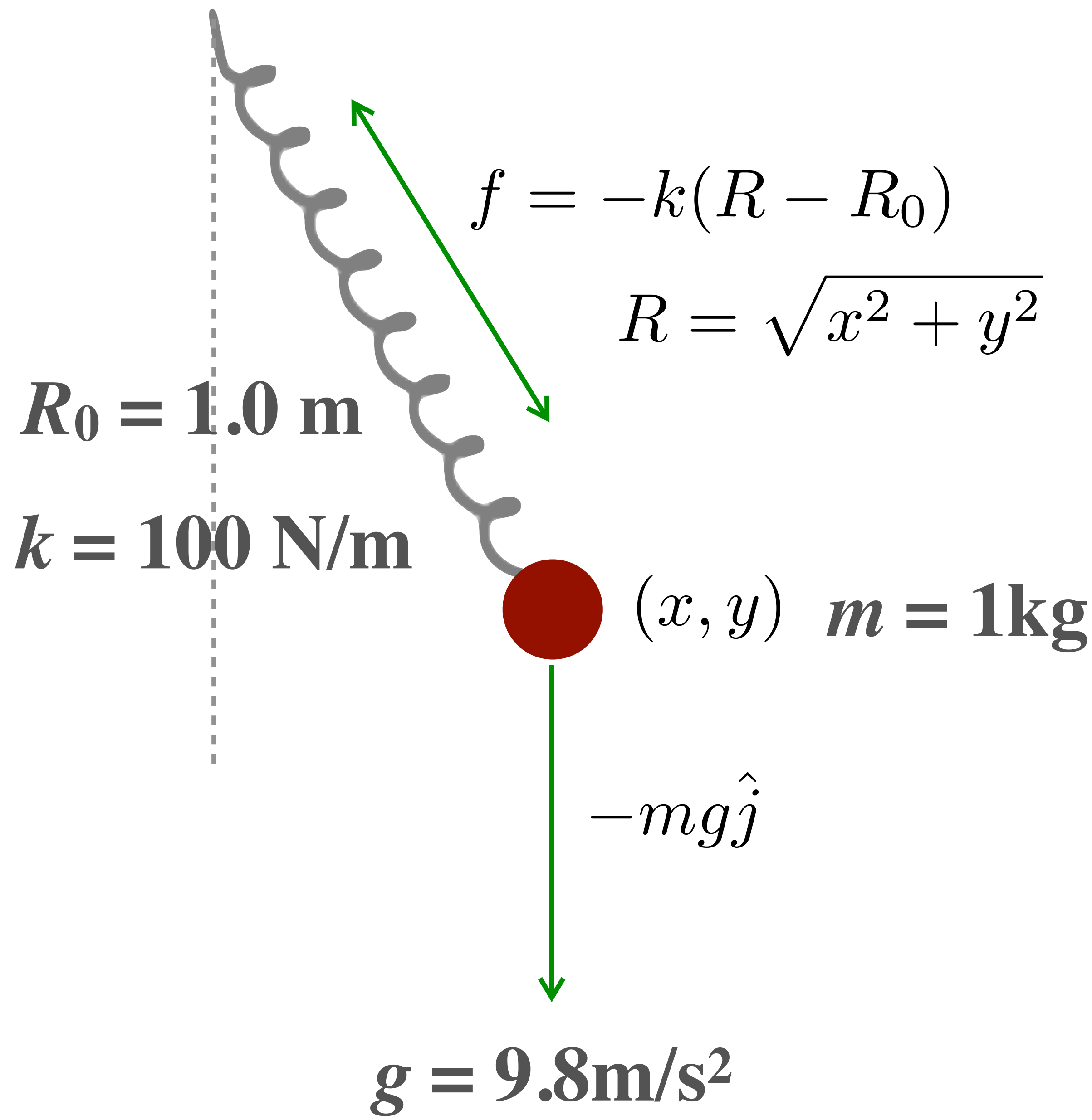
- What will happen if you given a critical initial condition to the preview simple pendulum example, e.g. $\theta = \pi$

  $\dot{\theta} = 0$

- It could be fun if you can try to record the angle versus time (this can be done by a small modification to l206-example-08.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**?

A SIMPLE VARIATION WITH SPRING

- Replace the “stick” with a spring:



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

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

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

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

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

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

A SIMPLE VARIATION WITH SPRING (II)

- Expand the equations in order to prepare the required ODE equations.
- Input array: [x, y, v_x, v_y]
- Output array:

$$\dot{x} = v_x$$

$$\dot{y} = v_y$$

$$\dot{v_x} = -k(R - R_0)\frac{x}{Rm}$$

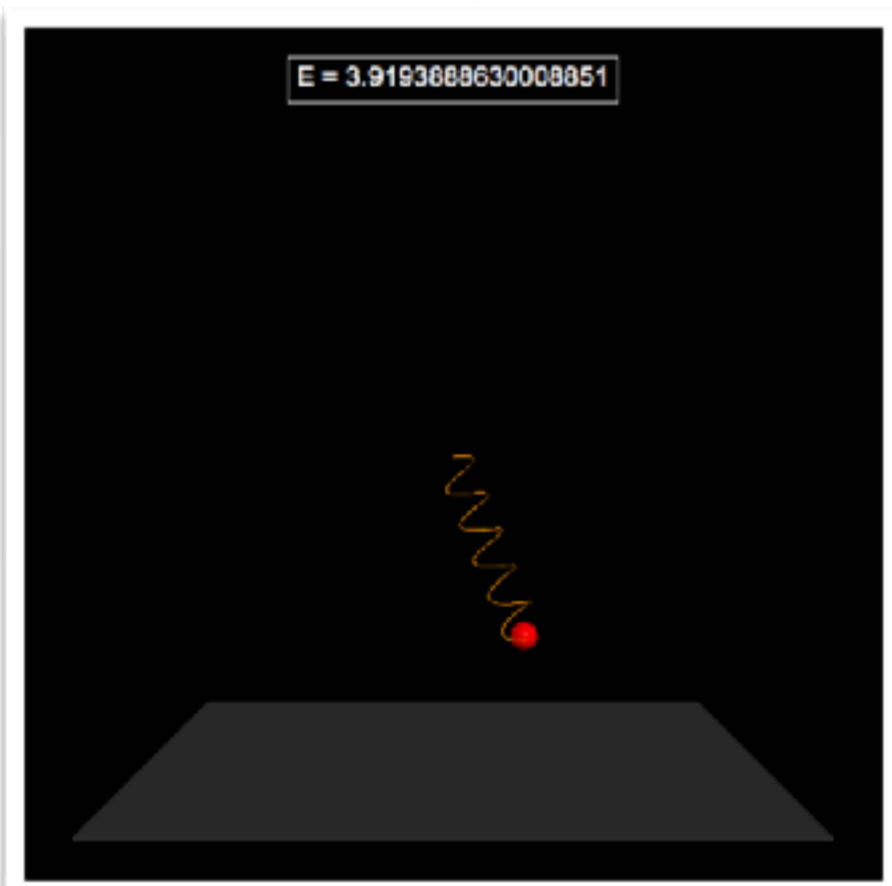
$$\dot{v_y} = -k(R - R_0)\frac{y}{Rm} - g$$

```
def f(t,y):
    bx, by = y[0], y[1]
    vx, vy = y[2], y[3]
    R = (bx**2+by**2)**0.5

    fs = -k*(R-R0)
    ax = fs*bx/R/m
    ay = fs*by/R/m - g
                         ⇓ output vector
    return np.array([vx,vy,ax,ay])
```

l206-example-09.py (partial)

A SIMPLE VARIATION WITH SPRING (III)

- The animation part is more-or-less the same as the previous example:

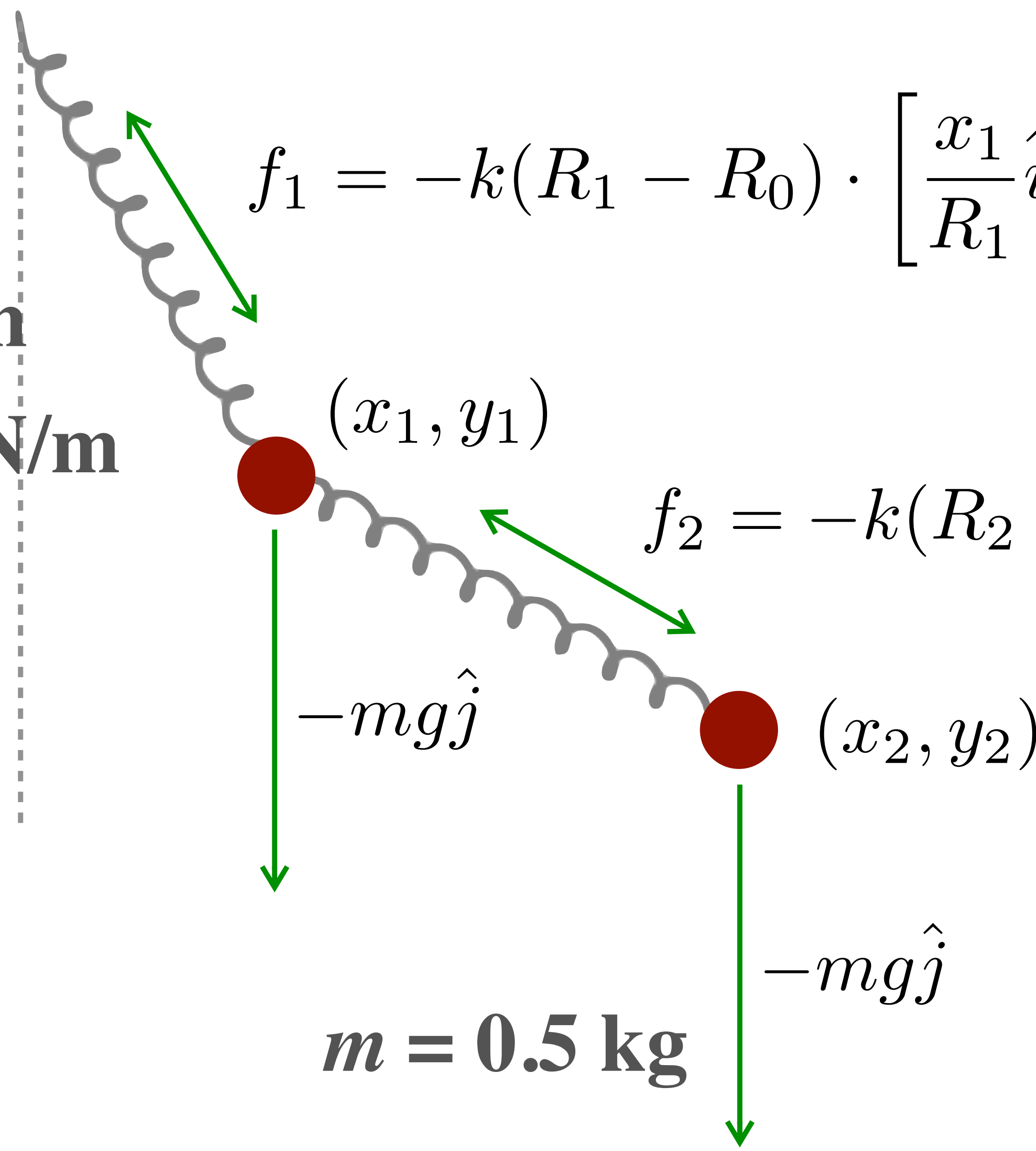



```
ball = sphere(...)
spring = helix(...)
txt = label(...)

while True:
    sol = solve_ivp(f,[t,t+0.040],y)
    y   = sol.y[:,-1]
    t   = sol.t[-1]

    bx, by = y[0], y[1]
    vx, vy = y[2], y[3]
    R = (bx**2+by**2)**0.5

    ball.pos.x = bx
    ball.pos.y = by
    spring.axis = ball.pos

    E = m*g*by + 0.5*m*(vx**2+vy**2)
      + 0.5*k*(R-R0)**2
    txt.text = 'E = %.16f' % E

    rate(1./0.040)
```

l206-example-09.py (partial)

FEW MORE EXAMPLES FOR YOUR AMUSEMENT

- A joint **two-spring-ball** system:



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

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

R$_0$ **= 0.5 m**

k **= 200 N/m**

m **= 0.5 kg**

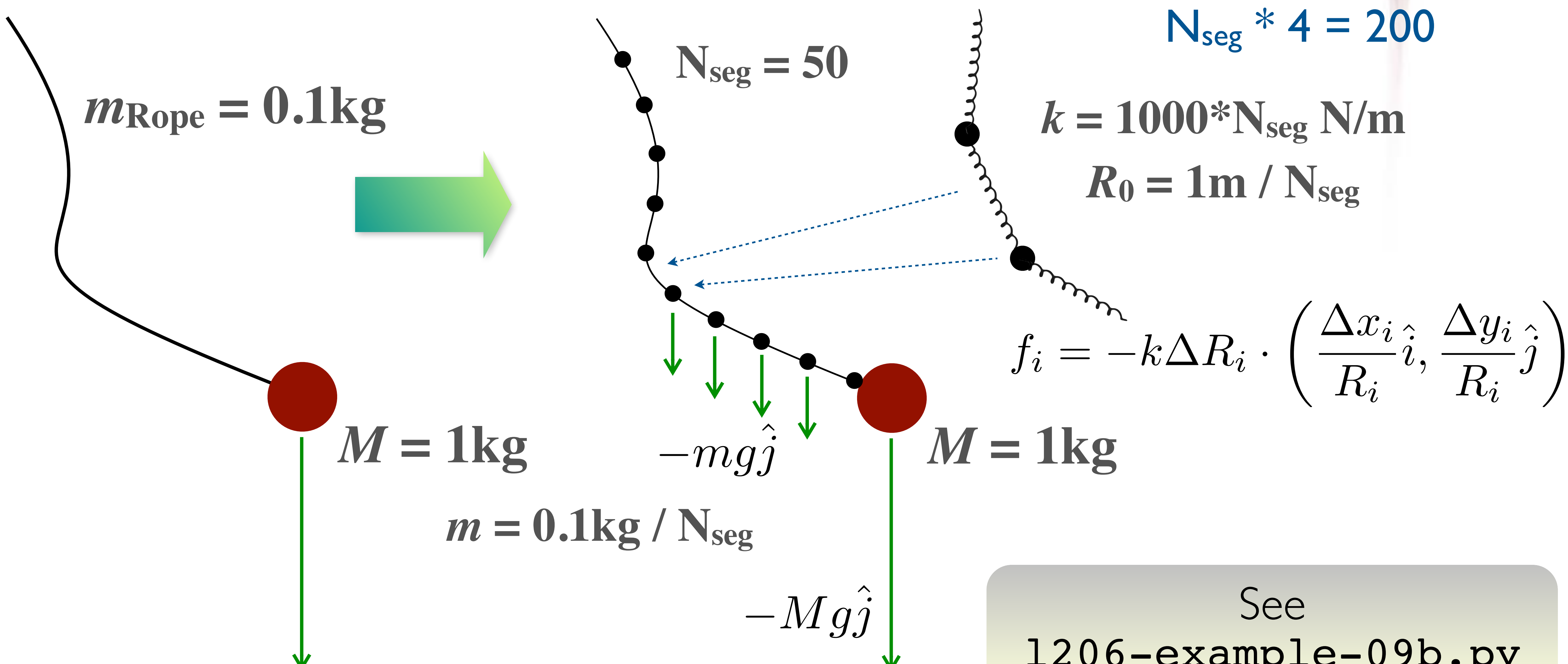
Need to solve 8 equations simultaneously

See `1206-example-09a.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: $N_{seg} * 4 = 200$

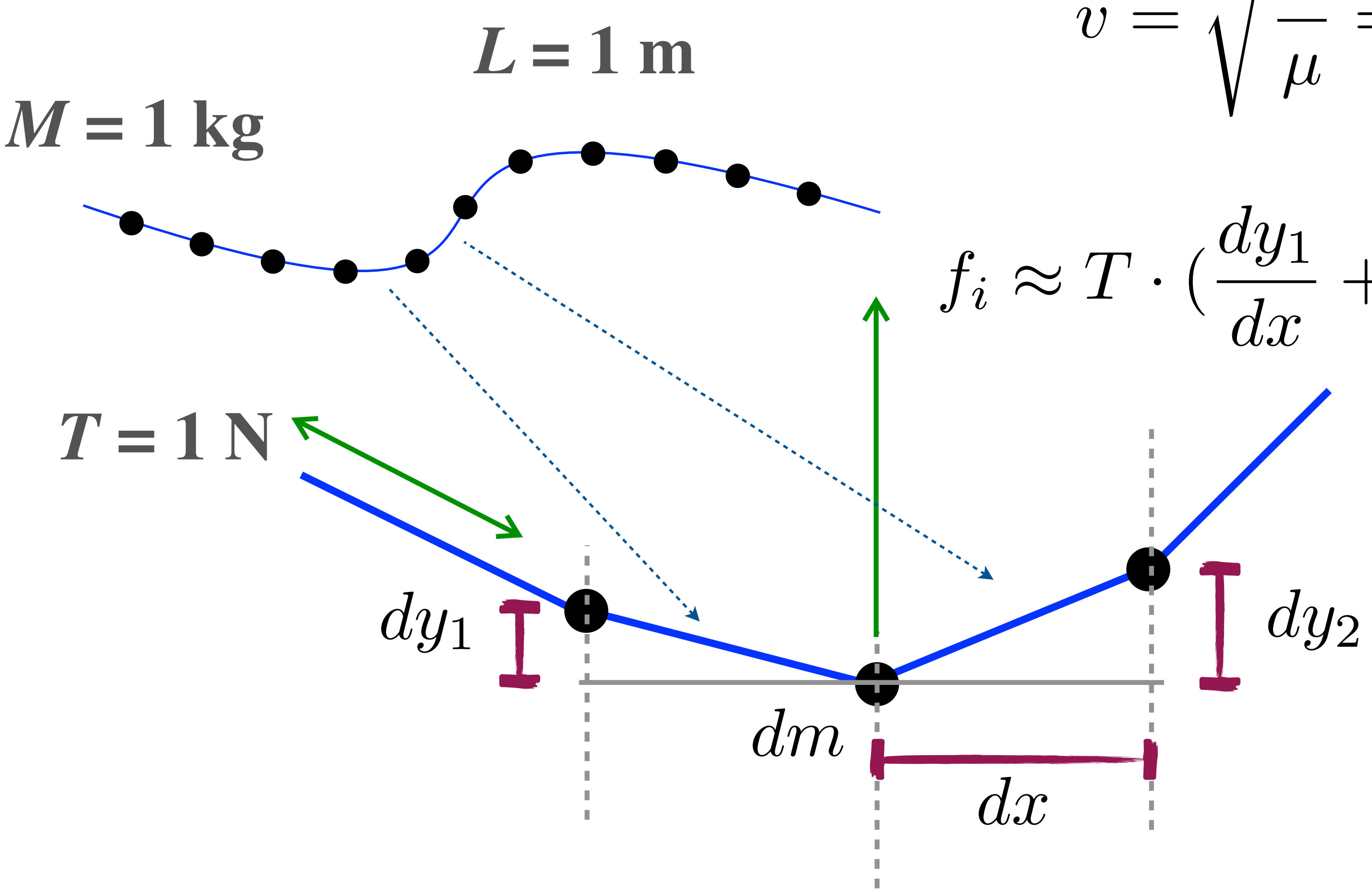


See
`1206-example-09b.py`

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.

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



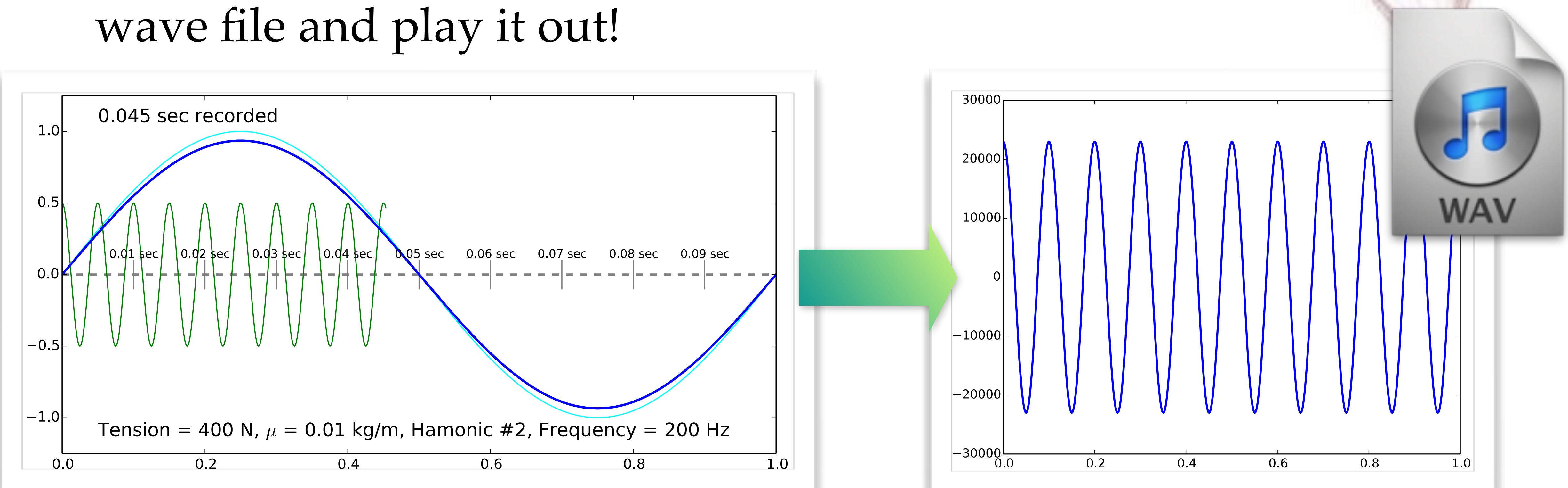
$$f_i \approx T \cdot \left(\frac{dy_1}{dx} + \frac{dy_2}{dx}\right)$$

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

See
`1206-example-10.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
`1206-example-10a.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

Left for your own study!

HANDS-ON SESSION

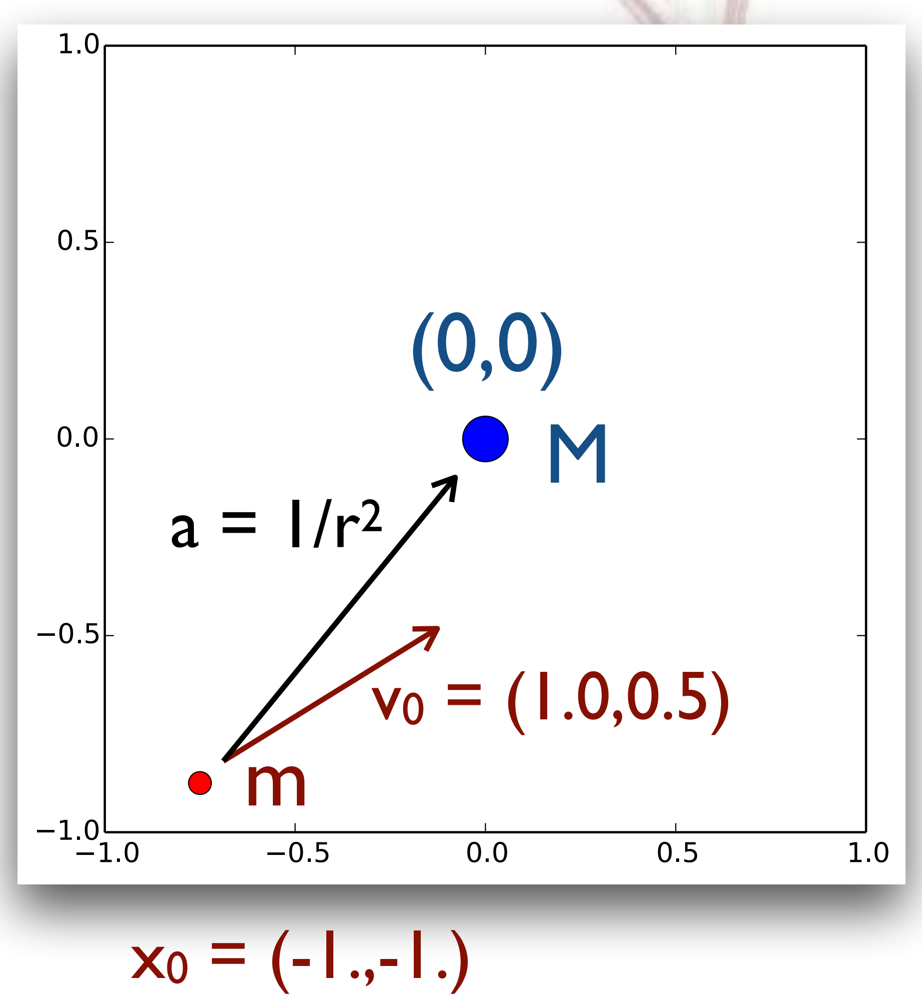
- **Practice 1:**
  Add some simple gravity to the system: there is a red star shooting toward the earth. Assuming the only acceleration between the earth and the red star is contributed by the gravitational force:

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

with G×M = 1. Thus:

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

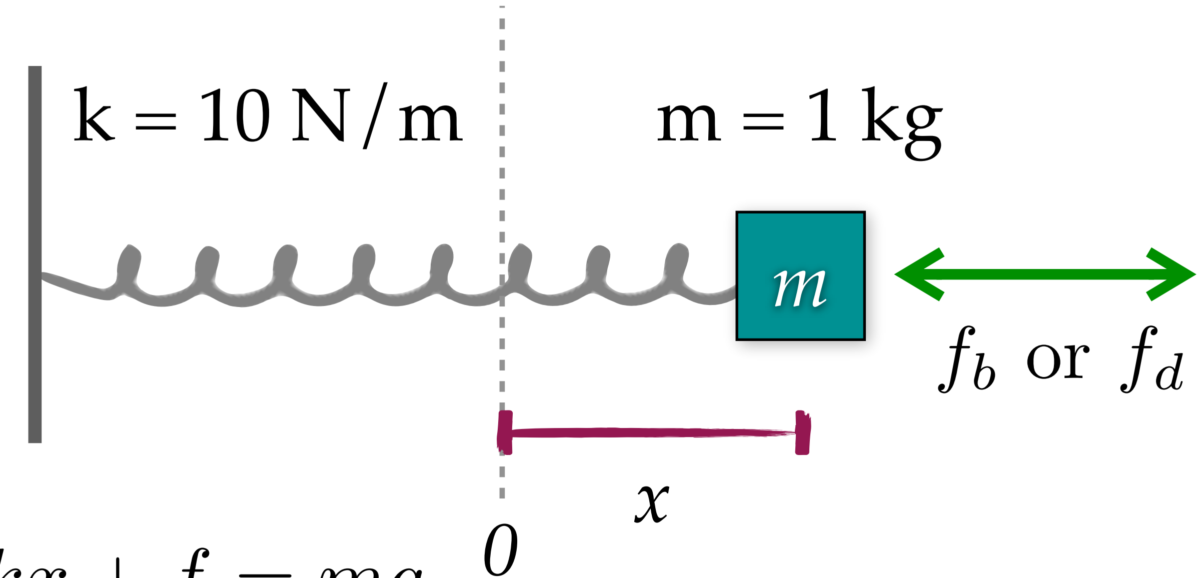
implement the code and produce the animation.



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**



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

or

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

or

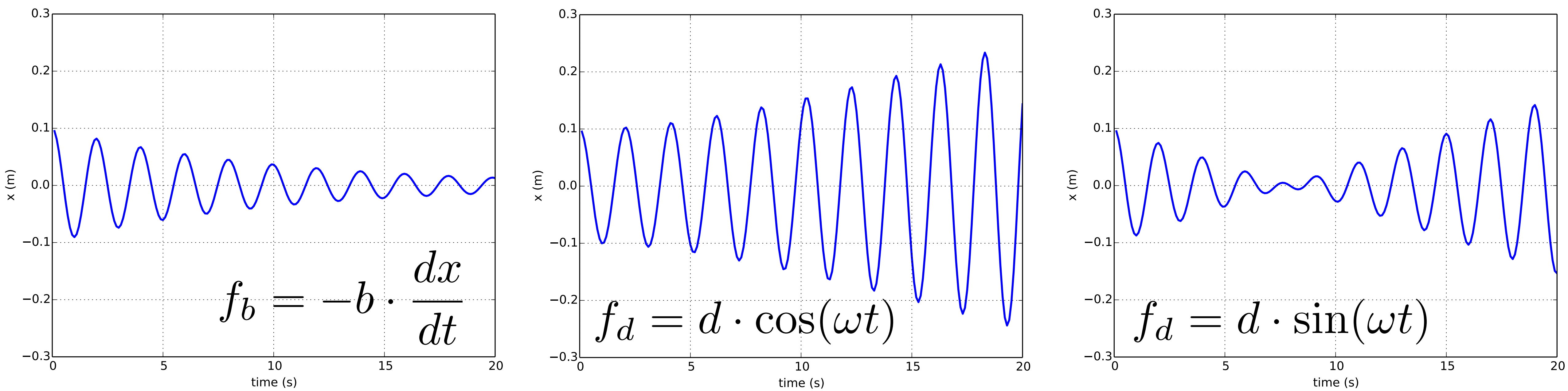
$$f_d = d \cdot \sin(\omega t) \qquad \omega = \pi\ \mathrm{rad/s}$$

$$F = -kx + f = ma$$

$$a = \frac{dv}{dt} = \frac{-kx + f}{m}$$

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.