Spring and Damper in Modelling

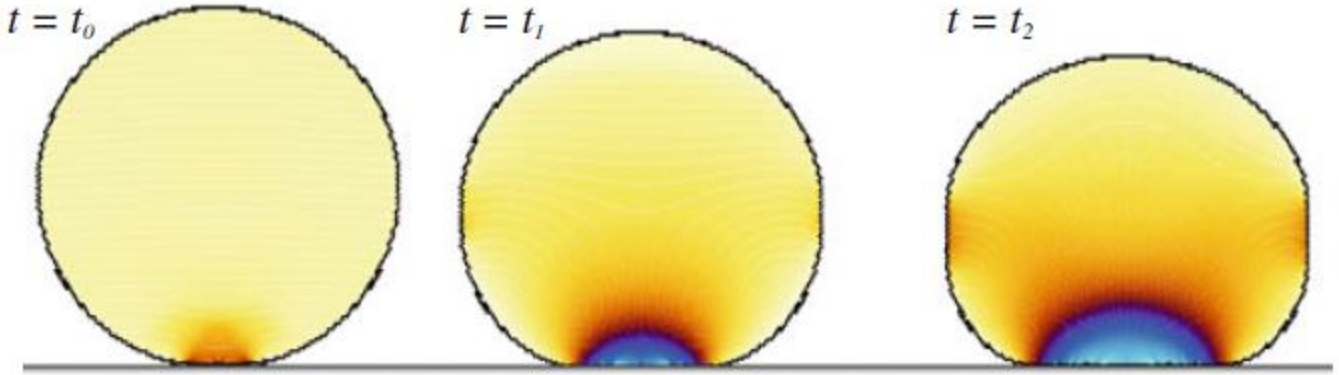
#8

Serdar ARITAN

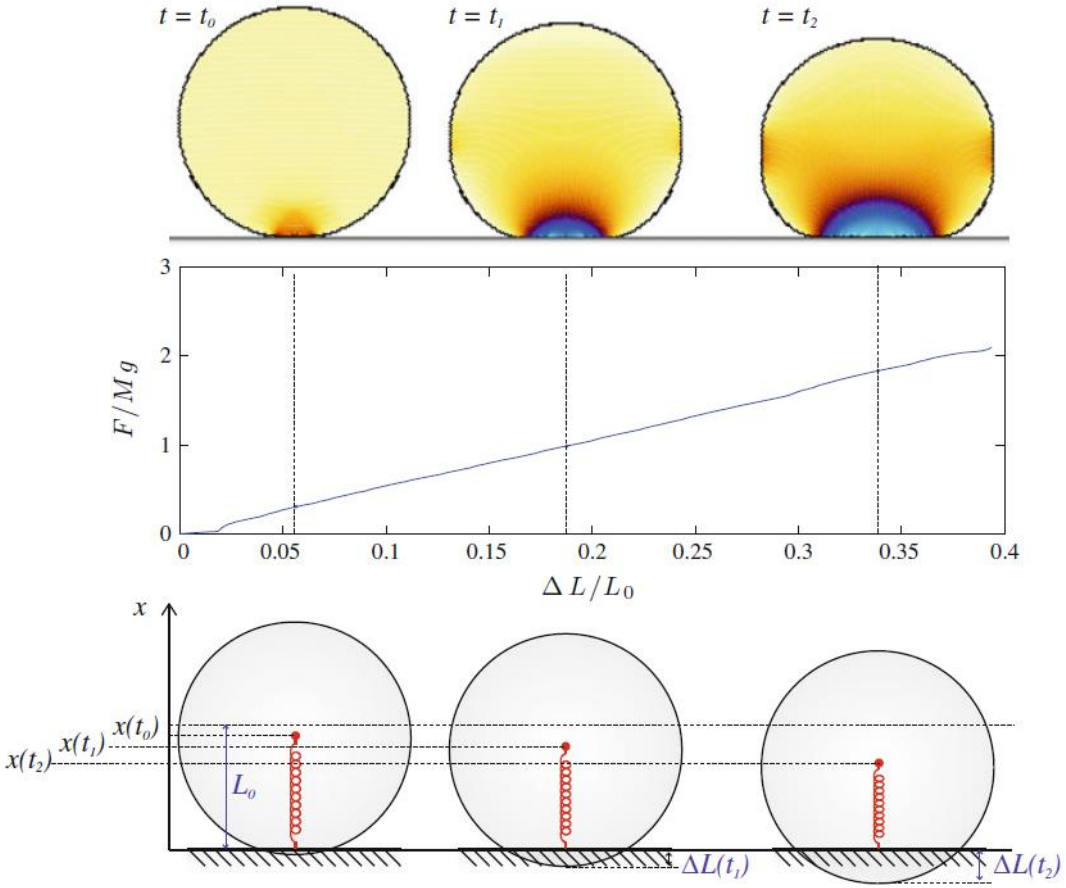
Biomechanics Research Group,
Faculty of Sports Sciences, and
Department of Computer Graphics
Hacettepe University, Ankara, Turkey

Contact Forces

The spring force was introduced as an approximative model for the force due to deformation. It is based on experimental evidence: We find the law by measuring the force as a function of the deformation. And the law is surprisingly versatile: We can use it as a model for almost any contact force between solid objects.



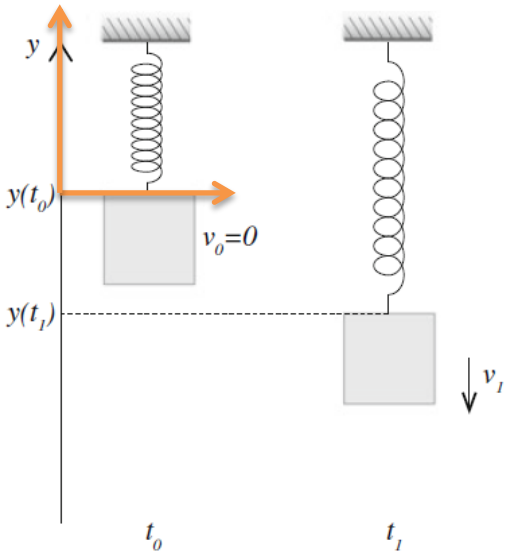
PHYSICS in COMPUTER ANIMATIONS and GAMES

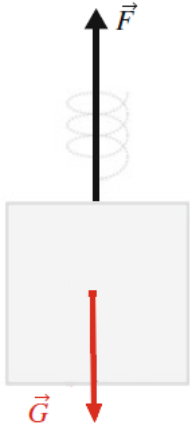


we use the **spring model** as an approximative model for the contact force. Generally, we do not know what the spring constant will be for such a model. We need either to measure the spring constant or to find the spring constant from a theoretical consideration based on for example elasticity theory.

Motion of a Hanging Block

A block of mass ***m*** = 1 kg is hanging from a spring with spring constant ***k*** = 100 N/m. The other end of the spring is attached to the ceiling. We apply the structured problem-solving approach to find the motion of the block after it is released.





$$F^{net} = \vec{F} - \vec{G}$$

$$\vec{F} = \pm k\Delta L$$

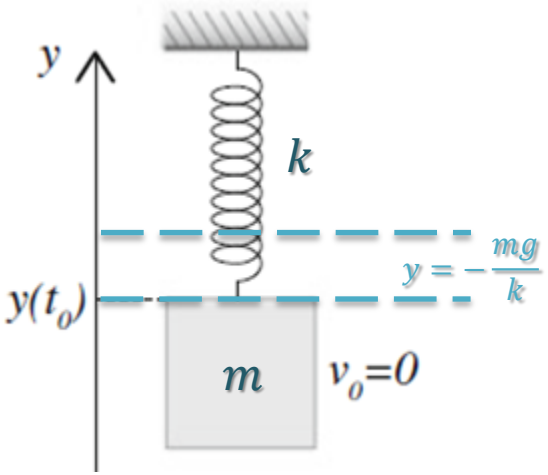
For simplicity, we place the coordinate system so that the spring is in equilibrium when y = 0 m

$$\vec{F}(y) = -ky\mathbf{j}$$

$$F^{net} = \vec{F} - \vec{G}$$

$$ma = -ky - mg$$

Equilibrium model: First, let us consider the equilibrium situation—where the block does not move when released. $a = 0$



$$ma = -ky - mg = 0$$

$$ma + ky + mg = 0$$

$$ky = -m(a + g)$$

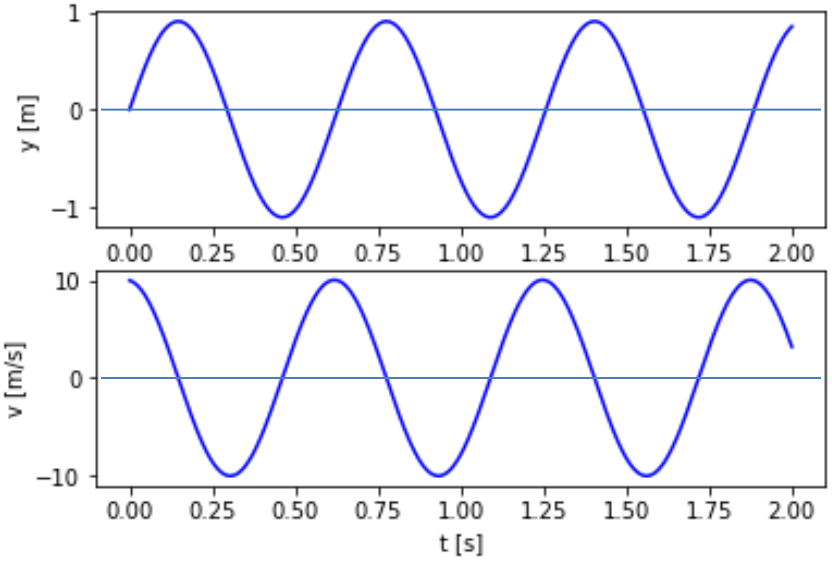
$$y = -\frac{mg}{k}$$

Numerical solution: We can find the motion of the block using an Euler scheme. Generally, we advice you to use **a fourth-order Runge-Kutta** method for oscillator problems, but we use Euler here to make the programming transparent.

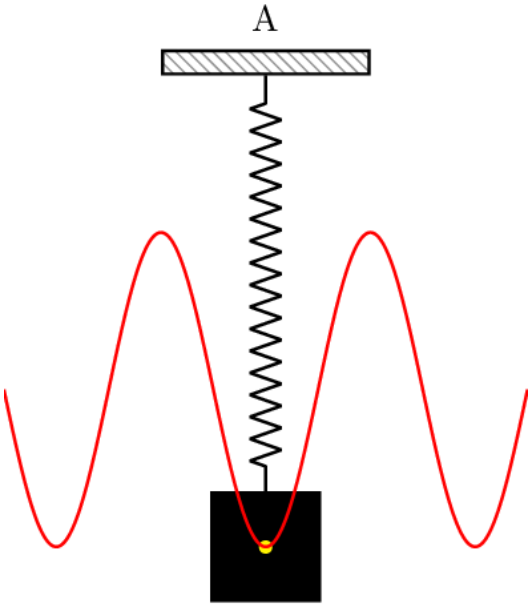
```
import numpy as np
import matplotlib.pyplot as plt
# Initialize
m = 1.0      # kg
k = 100.0    # N/m
v0 = 1.0     # m/s
time = 2.0   # s
g = 9.8      # m/s^2
# Numerical setup
dt = 0.0001 # s
n = int(round(time/dt))
t = np.zeros(n, float)
y = np.zeros(n, float)
v = np.zeros(n, float)
```

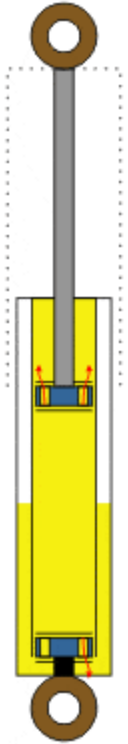
time step: please try 60 Hz (1/60)

```
# Initial values
y[0] = 0.0
v[0] = v0

# Simulation loop
for i in range(n-1):
    F = -k*y[i] - m*g
    a = F/m
    v[i+1] = v[i] + a*dt
    y[i+1] = y[i] + v[i+1]*dt
    t[i+1] = t[i] + dt

fig, (ax1, ax2) = plt.subplots(2, 1)
ax1.plot(t,y,'-b')
ax2.plot(t,v,'-b')
ax1.set_ylabel('y [m]')
ax2.set_ylabel('v [m/s]')
ax2.set_xlabel('t [s]')
plt.show()
```



Spring and Damper





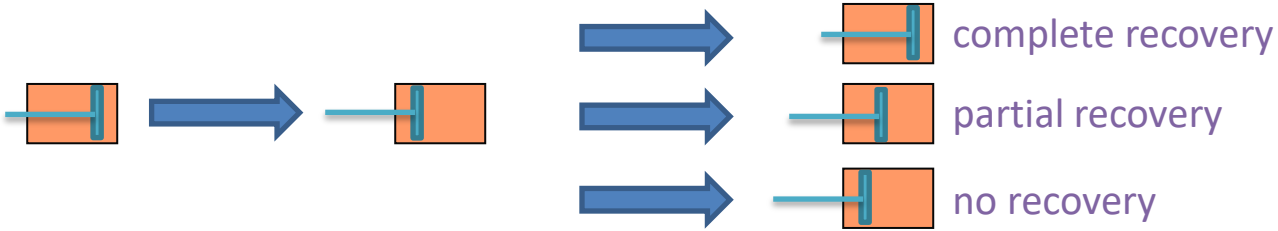
Spring and Damper

Spring: an ideal elastic element

Will immediately change and return to its original shape upon loading and unloading.

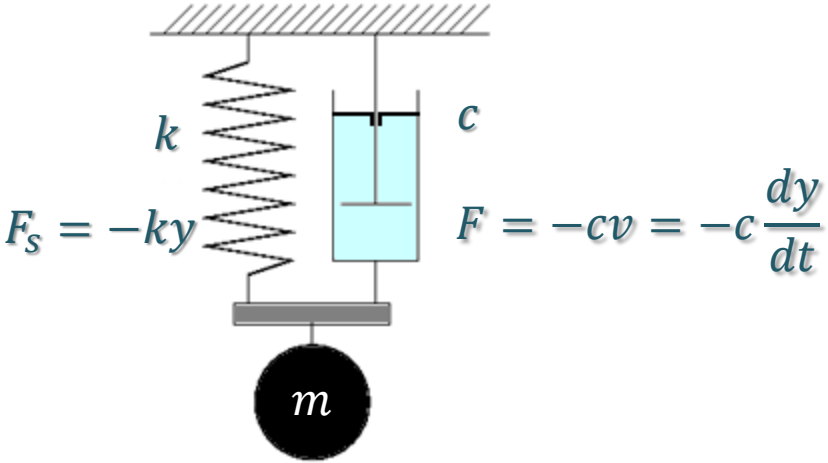
Damper: a viscous fluid

Will change its original shape upon loading, depending on time (and temperature).
May slowly return to or may not return to its original shape upon unloading.
partial or complete recovery



Mass–Spring–Damper (MSD)

An ideal mass–spring–damper system with mass **m**, spring constant **k**, and viscous damper of damping coefficient **c** is subject to an oscillatory force.



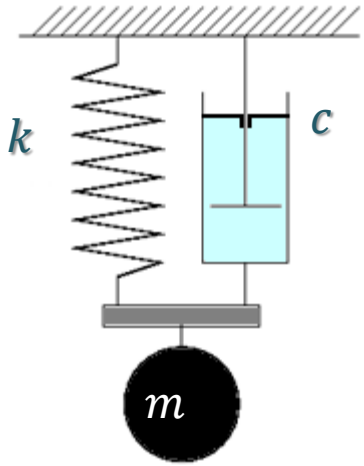
$$F^{net} = -ky + \left(-c\frac{dy}{dt}\right)$$

$$ma = -ky - c\frac{dy}{dt}$$

$$a = -\frac{k}{m}y - \frac{c}{m}\frac{dy}{dt}$$

$$\frac{d^2y}{dt^2} + \frac{c}{m}\frac{dy}{dt} + \frac{k}{m}y = 0$$

2. Order Differential Equation



$$\zeta = \frac{c}{2\sqrt{km}}$$ the damping ratio

$$\omega_0 = \sqrt{\frac{k}{m}}$$ the natural frequency of the system

$$\frac{d^2y}{dt^2} + 2\zeta\omega_0\frac{dy}{dt} + \omega_0^2 y = 0$$

The first parameter, ω_0, is called the (undamped) natural frequency of the system. The second parameter, ζ, is called the damping ratio. The natural frequency represents an angular frequency, expressed in radians per second. The damping ratio is a dimensionless quantity.

By choosing appropriate parameters we can "tune" the behavior of the system

We can classify the system into three distinct categories.

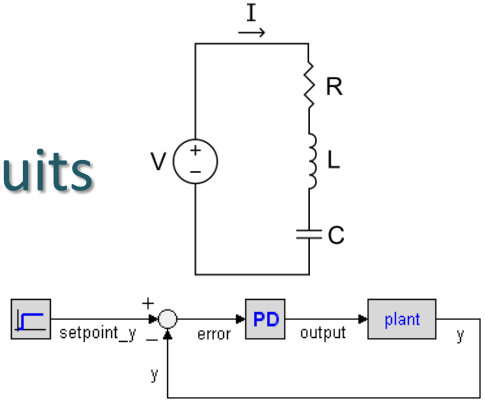
1. Under-damped: $\zeta < 1$. In this case the system oscillates with a frequency equal to $\omega_d = \omega_0\sqrt{1-\zeta^2}$

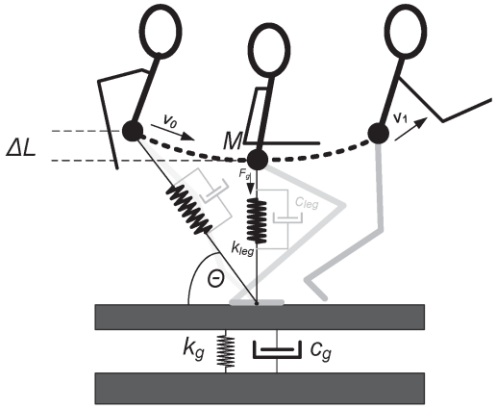
2. Over-damped: $\zeta > 1$. The system slowly returns to equilibrium

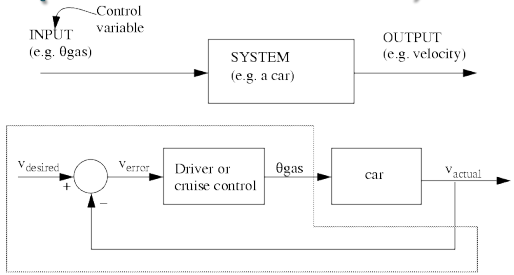
3. Critically Damped: $\zeta = 1$. The system returns to equilibrium

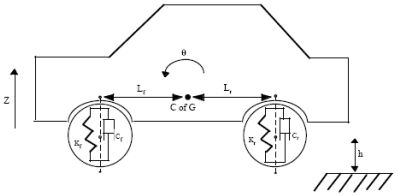
MSD-like Systems are found throughout engineering

- Human Running
- Circuit Design: RLC circuits
- Robotics (PD control)
- Automotive (Cruise control, car suspensions etc.)









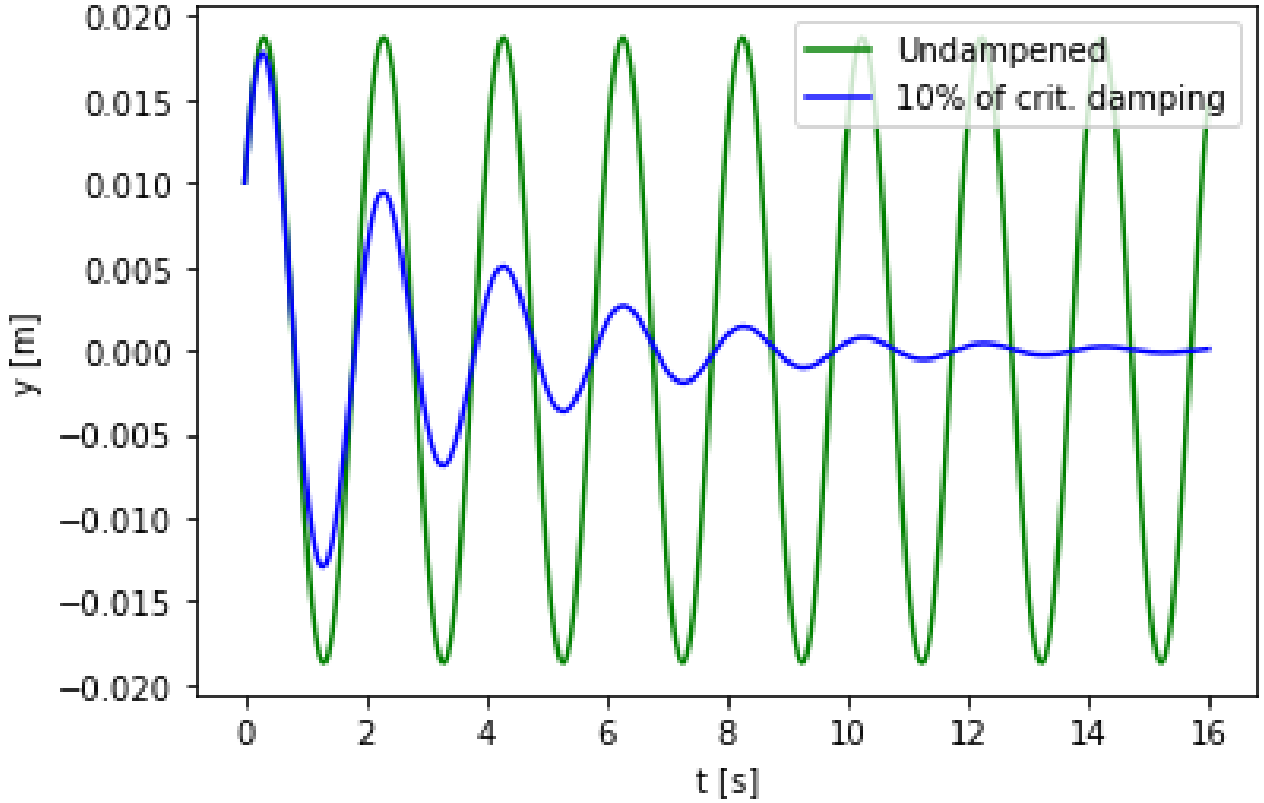
PHYSICS in COMPUTER ANIMATIONS and GAMES

```
import numpy as np
import matplotlib.pyplot as plt
# Initialize
timeFinal= 16.0    # Simulation time in seconds
steps = 10000      # Number of steps
dt = timeFinal/steps       # Step length => 0.0016 s
t = np.linspace(0, timeFinal, steps+1)
# Creates an array with steps+1 values from 0 to timeFinal
# Allocating arrays for velocity and position
v = np.zeros(steps+1)
y = np.zeros(steps+1)
# Setting constants and initial values for vel. and pos.
k = 0.1
m = 0.01
v0 = 0.05
y0 = 0.01
freqNatural = (k/m)**0.5
c = 0.00      #Undampened
v[0] = v0     #Sets the initial velocity
y[0] = y0     #Sets the initial position
```

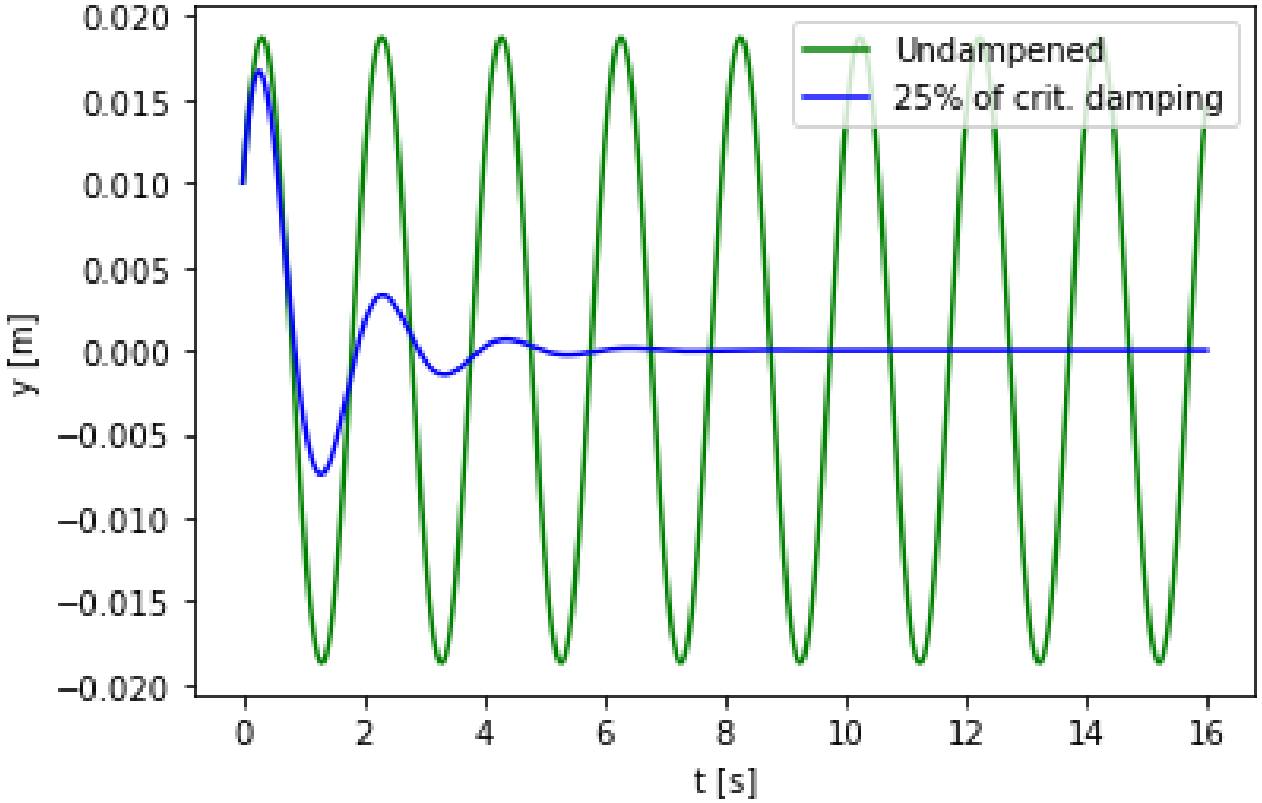
PHYSICS in COMPUTER ANIMATIONS and GAMES

```
# Simulation loop
# Numerical solution using Euler's
# v'(t) = -(k/m)*x(t) - (c/m)*v(t)
# x'(t) = v(t)
for i in range(0, steps):
    v[i+1] = (-k/m)*dt*y[i] + v[i]*(1-dt*c/m)
    y[i+1] = y[i] + v[i+1]*dt
fig, ax = plt.subplots()
ax.plot(t, y,'-g', label='Undampened')
# Damping set to 10% of critical damping
c = (2*(k*m)**0.5)*0.10
for i in range(0, steps):
    v[i+1] = (-k/m)*dt*y[i] + v[i]*(1-dt*c/m)
    y[i+1] = y[i] + v[i+1]*dt

ax.plot(t, y, 'b-', label = '10% of crit. damping')
ax.xlabel('t [s]')
ax.set_ylabel('y [m]')
ax.legend(loc = 'upper right')
plt.show()
```

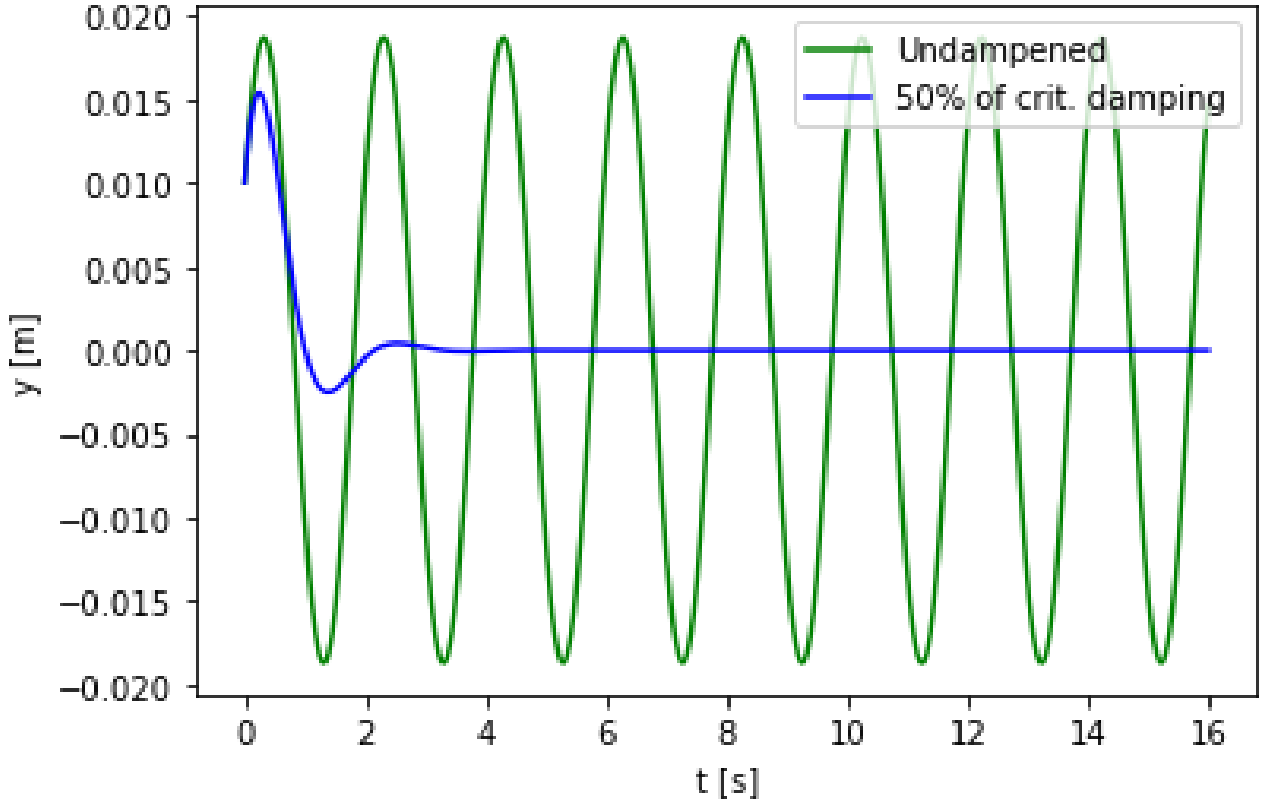
```
c = (2*(k*m)**0.5)*0.10
```


Undampened
10% of crit. damping
y [m]
t [s]
0.020
0.015
0.010
0.005
0.000
−0.005
−0.010
−0.015
−0.020
0
2
4
6
8
10
12
14
16

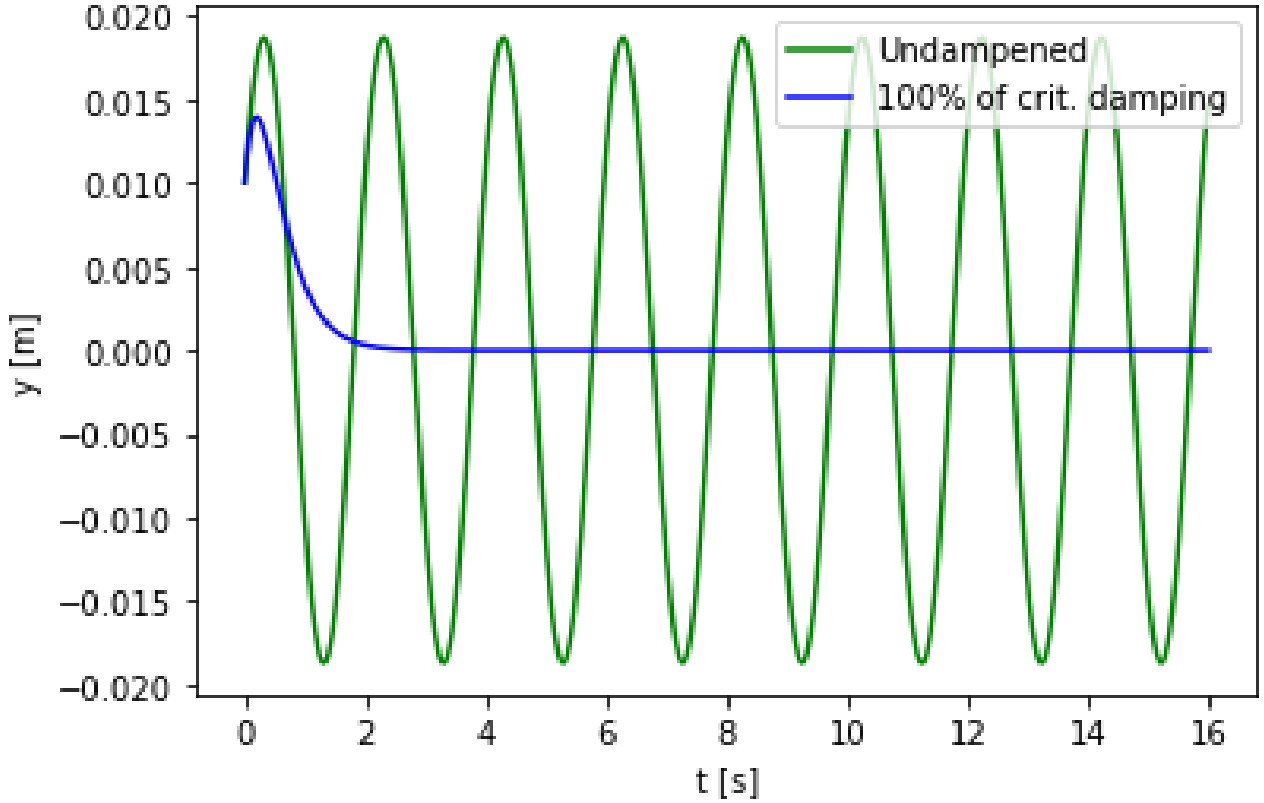
```
c = (2*(k*m)**0.5)*0.25
```

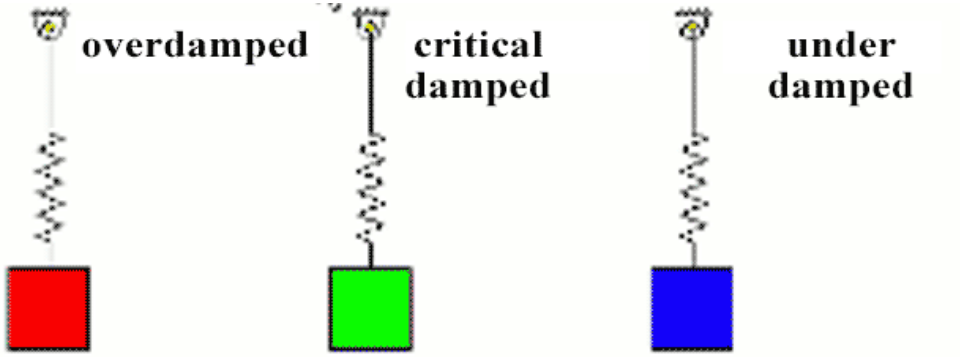

Undampened
25% of crit. damping
y [m]
t [s]
0.020
0.015
0.010
0.005
0.000
−0.005
−0.010
−0.015
−0.020
0
2
4
6
8
10
12
14
16

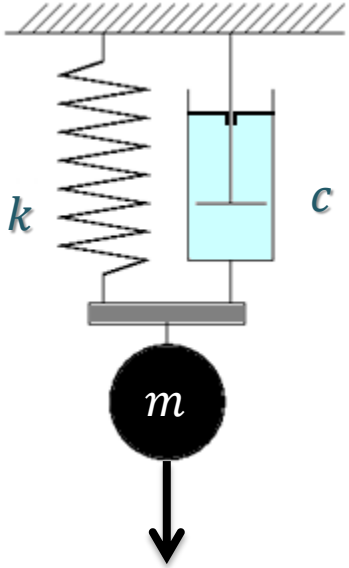
```
c = (2*(k*m)**0.5)*0.50
```


Undampened
50% of crit. damping
y [m]
t [s]
0.020
0.015
0.010
0.005
0.000
−0.005
−0.010
−0.015
−0.020
0
2
4
6
8
10
12
14
16

```
c = (2*(k*m)**0.5)*1.00
```

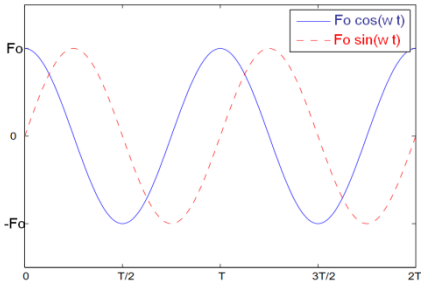

Undampened
100% of crit. damping
y [m]
t [s]
0.020
0.015
0.010
0.005
0.000
−0.005
−0.010
−0.015
−0.020
0
2
4
6
8
10
12
14
16


overdamped
critical damped
under damped



External Forcing models the behavior of a system which has a time varying force acting on it. An example might be a suspended bridge subjected to wind loading.

$$F(t) = F_0 \sin \omega t$$



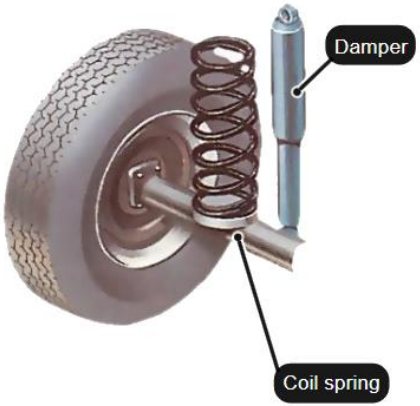
External Forcing

Natural frequency is the frequency at which a system tends to oscillate in the absence of any driving **or damping force**.

Natural frequency depends on the elasticity and shape of the object.

The natural frequency of the smaller bell is higher than that of the big bell, and it rings at a higher pitch.

The natural frequency of a trombone can be modified by changing the length of the air column inside the metal tube.

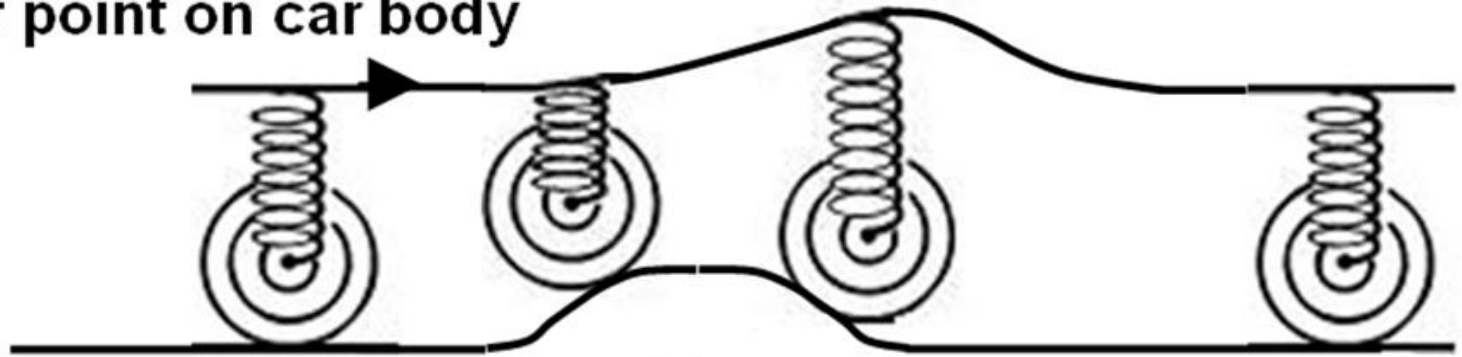


$$F(t) = F_0 \cos \omega t$$

$$F_0 = 0.01$$

$$\omega = \sqrt{k/m}$$

path of point on car body



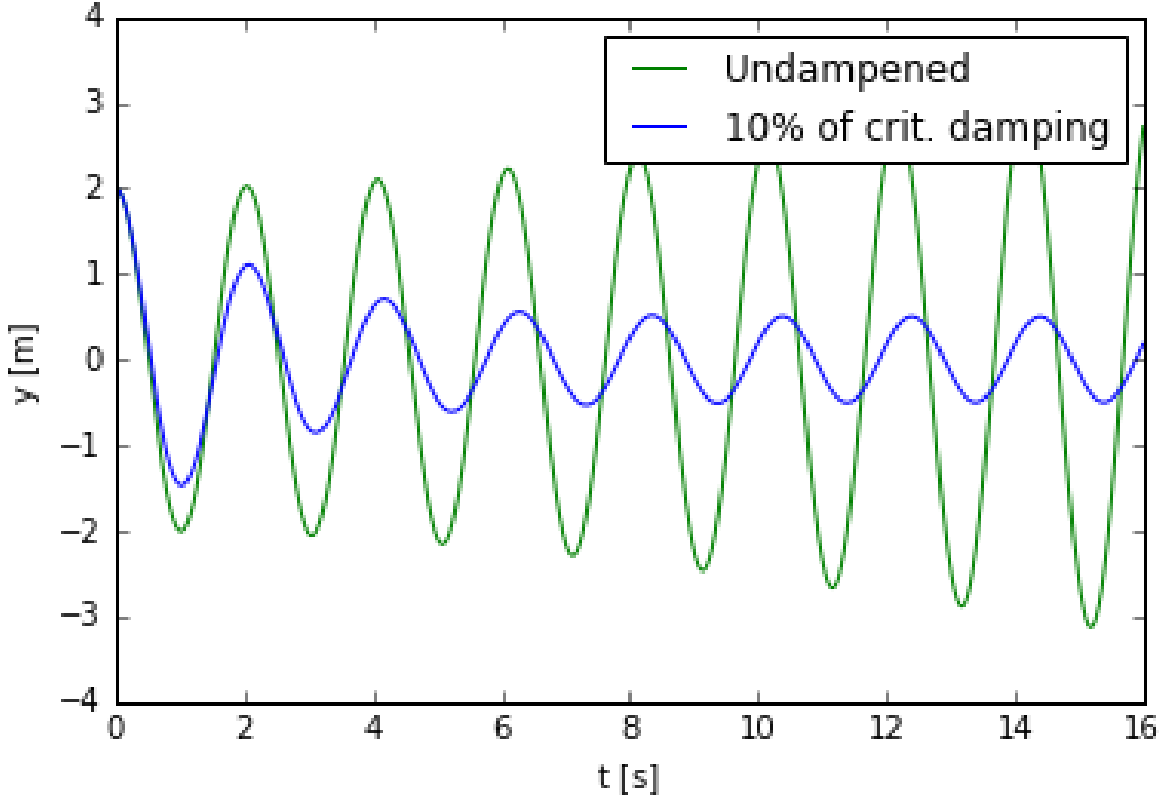
road hump

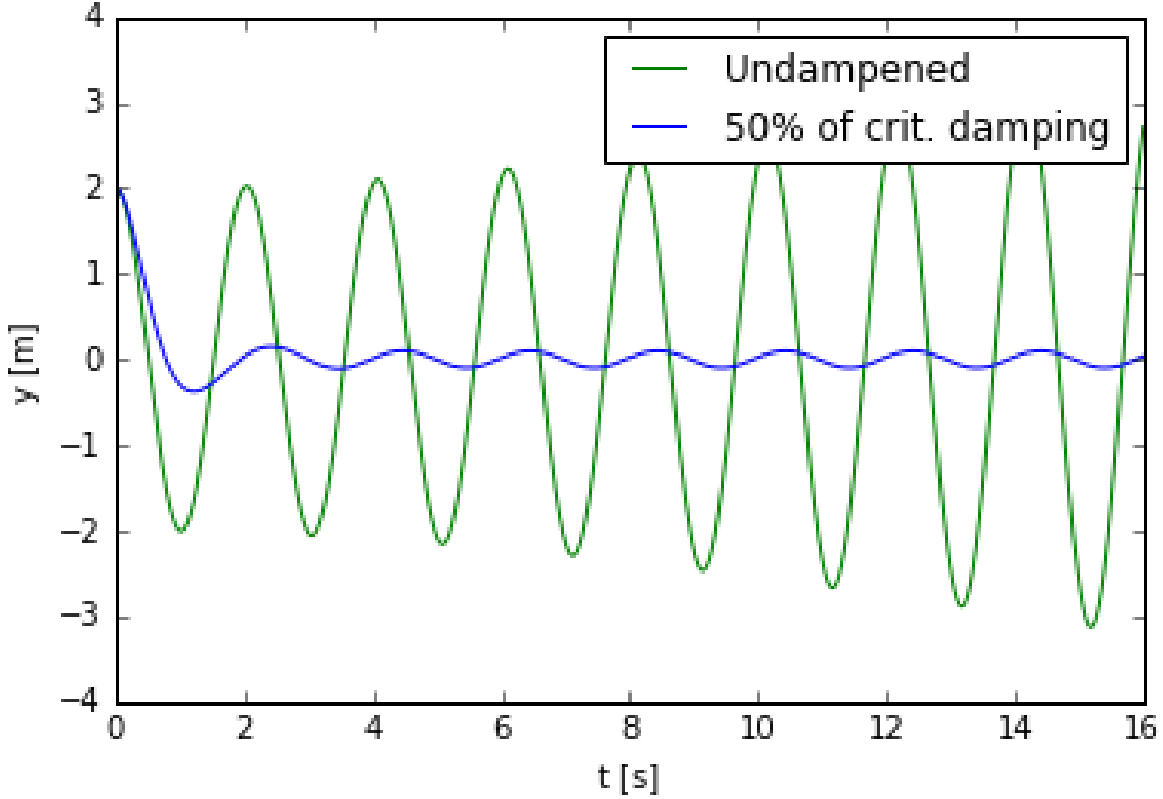
PHYSICS in COMPUTER ANIMATIONS and GAMES

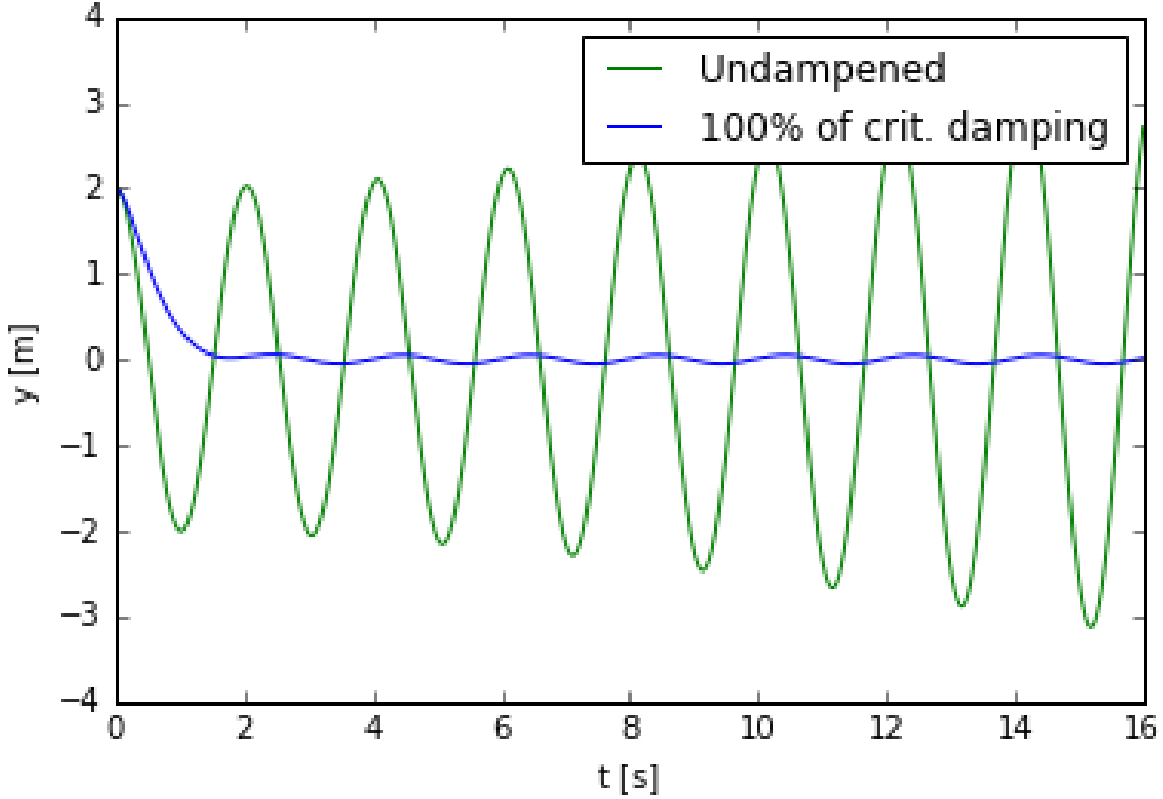
```
import numpy as np
import matplotlib.pyplot as plt
# Initialize
timeFinal= 16.0    # Simulation time in seconds
steps = 10000      # Number of steps
dt = timeFinal/steps       # Step length
t = np.linspace(0, timeFinal, steps+1)
# Creates an array with steps+1 values from 0 to timeFinal
v = np.zeros(steps+1)
y = np.zeros(steps+1)
# Setting constants and initial values for vel. and pos.
k = 0.1
m = 0.01
v0 = 0.01
y0 = 2.00
freqNatural = (k/m)**0.5
c = 0.00      #Undampened
F0 = 0.010
Wd = freqNatural
v[0] = v0     #Sets the initial velocity
y[0] = y0     #Sets the initial position
```

PHYSICS in COMPUTER ANIMATIONS and GAMES

```
# Simulation loop
for i in range(0, steps):
    v[i+1] = (-k/m)*y[i]*dt + v[i]*(1-dt*c/m) + F0/m*np.cos(Wd*i*dt)*dt
    y[i+1] = y[i] + v[i+1]*dt

fig, ax = plt.subplots()
ax.plot(t, y,'-g', label='Undampened')
# Damping set to 10% of critical damping
c = (2*(k*m)**0.5)*0.10
for i in range(0, steps):
    v[i+1] = (-k/m)*y[i]*dt + v[i]*(1-dt*c/m) + F0/m*np.cos(Wd*i*dt)*dt
    y[i+1] = y[i] + v[i+1]*dt

ax.plot(t, y, 'b-', label = '10% of crit. damping')
ax.xlabel('t [s]')
ax.ylabel('y [m]')
ax.legend(loc = 'upper right')
plt.show()
```


Undampened
10% of crit. damping
y [m]
t [s]
4
3
2
1
0
−1
−2
−3
−4
0
2
4
6
8
10
12
14
16


Undampened
50% of crit. damping
y [m]
t [s]
4
3
2
1
0
−1
−2
−3
−4
0
2
4
6
8
10
12
14
16


Undampened
100% of crit. damping
y [m]
t [s]
4
3
2
1
0
−1
−2
−3
−4
0
2
4
6
8
10
12
14
16

Viscoelastic Materials

- Viscosity = The resistance to motion of a liquid
- Viscoelastic materials

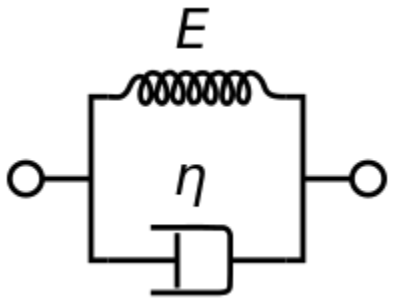
  Properties = Elastic solid + Fluid

  = Materials that have mechanical properties dependent on time (loading rate or strain rate) and temperature

Material Modelling

- Kelvin–Voigt material

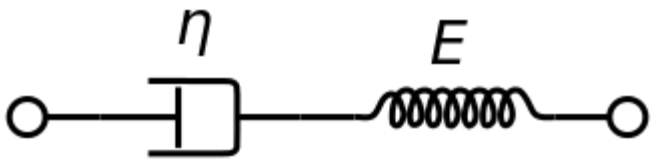
  A Kelvin–Voigt material, also called a Voigt material, is a viscoelastic material having the properties both of elasticity and viscosity. It is named after the British physicist and engineer Lord Kelvin and after German physicist Woldemar Voigt. the Kelvin–Voigt model does not describe stress relaxation.



Material Modelling

- Maxwell material

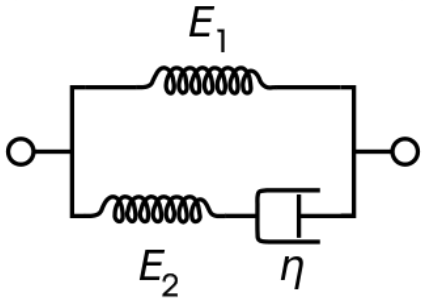
  A Maxwell material is a viscoelastic material having the properties both of elasticity and viscosity. It is named for James Clerk Maxwell who proposed the model in 1867. It is also known as a Maxwell fluid. The Maxwell model does not describe creep or recovery.



Material Modelling

- Standard linear solid material

  The standard linear solid (SLS) model is a method of modeling the behavior of a viscoelastic material using a linear combination of springs and dashpots to represent elastic and viscous components, respectively. SLS is the simplest model that predicts creep or recovery, and stress relaxation phenomena.



scipy.integrate.solve_ivp

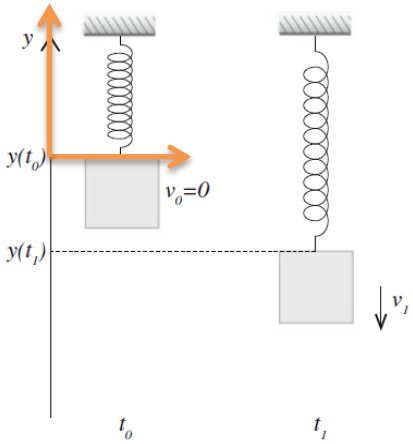
scipy.integrate.solve_ivp(*fun, t_span, y0, method='RK45', t_eval=None, dense_output=False, events=None, vectorized=False, args=None, **options*) [source]

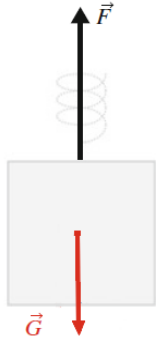
Solve an initial value problem for a system of ODEs.

This function numerically integrates a system of ordinary differential equations given an initial value:

```
# Simulation loop
for i in range(n-1):
    F = -k*y[i] - m*g
    a = F/m
    v[i+1] = v[i] + a*dt
    y[i+1] = y[i] + v[i+1]*dt
    t[i+1] = t[i] + dt
```

Euler integration





$$F^{net} = \vec{F} - \vec{G}$$

$$ma = -ky - mg$$

$$a = -ky/m - g$$

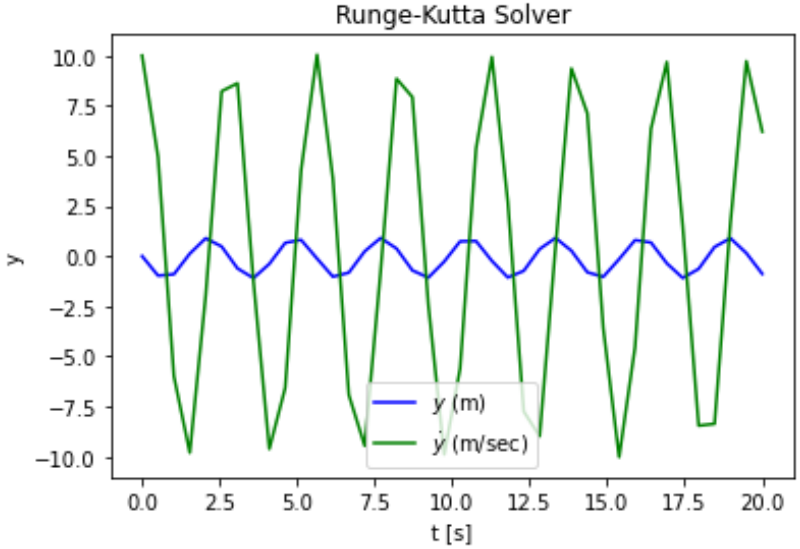
```
def MassSpring(t, state, m, k, g):
  # unpack the state vector
  x = state[0]
  xd = state[1]
  # compute acceleration xdd
  F = -k*x - m*g
  xdd = F/m    # xdd = ((k*x)/m) - g
  #return the two state derivatives
  return [xd, xdd]
```

PHYSICS in COMPUTER ANIMATIONS and GAMES

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp

# simulation parameters
stime = 20.0 # time [s]
m = 1.0      # mass [kg]
k = 100.0    # spring stiffness [N/m]
v0 = 10.0    # initial velocity [m/s]
g = 9.8      # gravity [m/s^2]
dt = 0.5     # time step [s]
n = int(round(stime/dt))
tspan = np.linspace(0.0, stime, n)
```

PHYSICS in COMPUTER ANIMATIONS and GAMES

```
# ODE function
def MassSpring(t, state, m, k, g):
    x = state[0]
    xd = state[1]
    # compute acceleration xdd
    F = -k*x - m*g
    xdd = F/m      # xdd = ((k*x)/m) - g
    # return the two state derivatives
    return [xd, xdd]


sol = solve_ivp(MassSpring,[tspan[0],tspan[-1]],
                                0, v0],
                                method = "RK45",
                                t_eval=tspan, args=(m, k, g))
```




```
# %% Plot states
fig, ax = plt.subplots()
ax.plot(sol.t, sol.y[0],'-b')
ax.plot(sol.t, sol.y[1],'-g')
ax.set_xlabel('t [s]')
ax.set_ylabel('y')
plt.title('Runge-Kutta Solver')
ax.legend(('$y$ (m)', '$\dot{y}$ (m/sec)'))
plt.show()
```

‘RK45’ (default): Explicit Runge-Kutta method of order 5(4). The error is controlled assuming accuracy of the fourth-order method, but steps are taken using the fifth-order accurate formula.

‘RK23’: Explicit Runge-Kutta method of order 3(2). The error is controlled assuming accuracy of the second-order method, but steps are taken using the third-order accurate formula.

‘DOP853’: Explicit Runge-Kutta method of order 8. Python implementation of the “DOP853” algorithm originally written in Fortran.

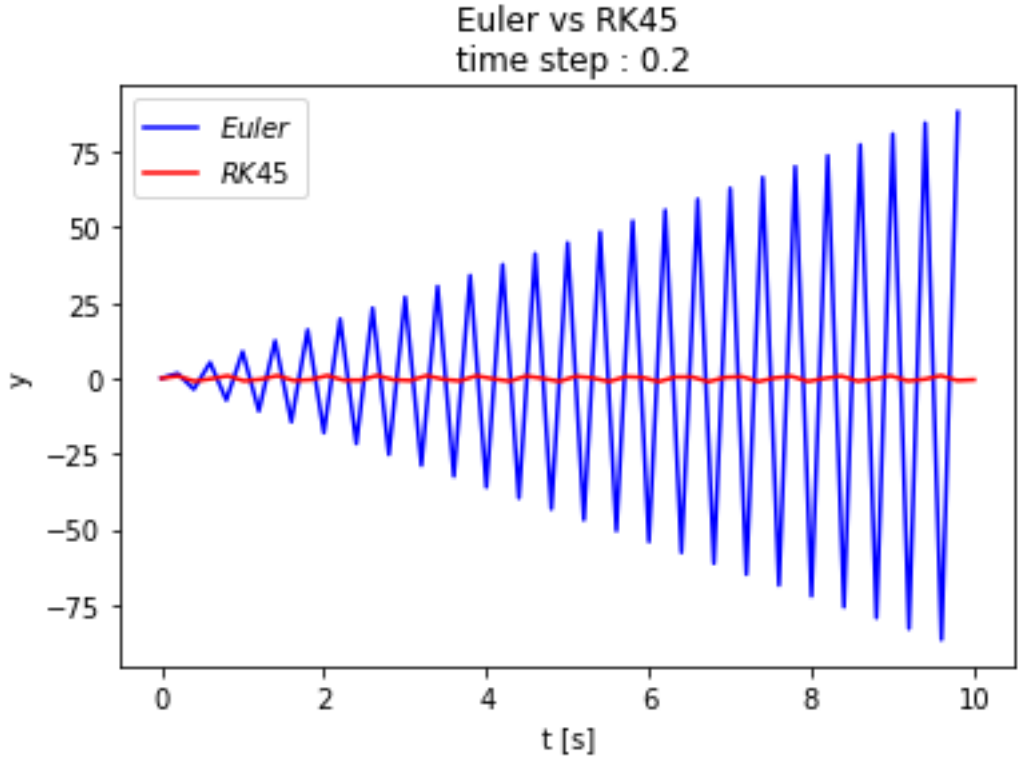
‘Radau’: Implicit Runge-Kutta method of the Radau IIA family of order 5. The error is controlled with a third-order accurate embedded formula.

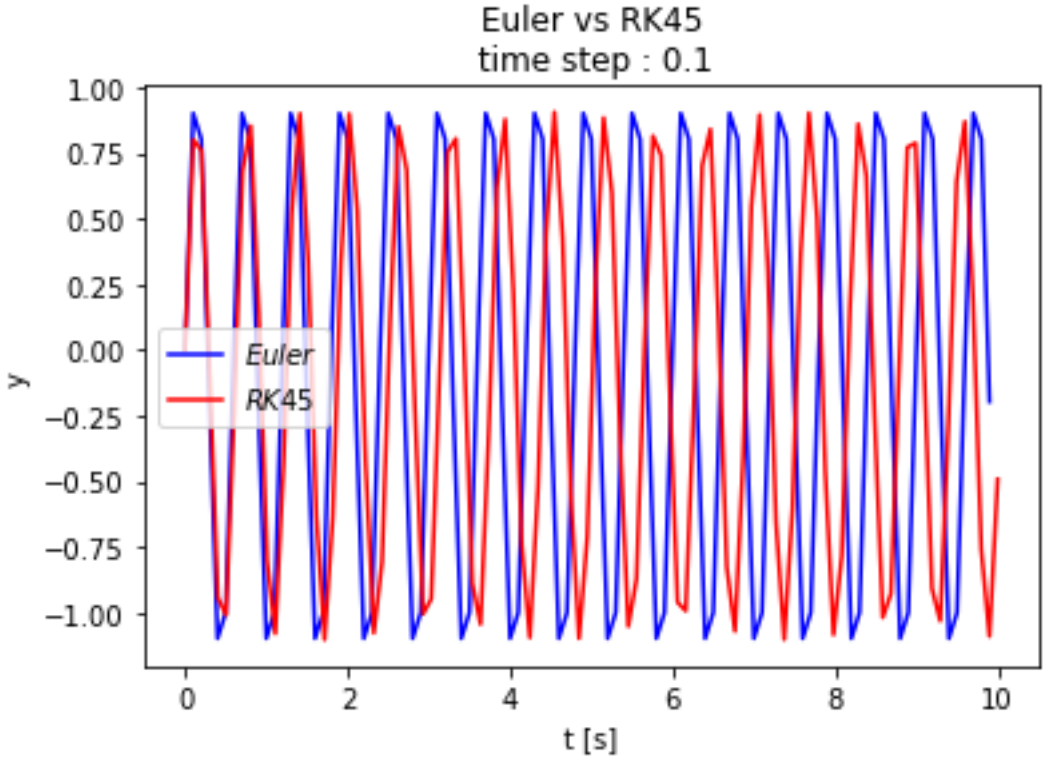
‘BDF’: Implicit multi-step variable-order (1 to 5) method based on a backward differentiation formula for the derivative approximation.

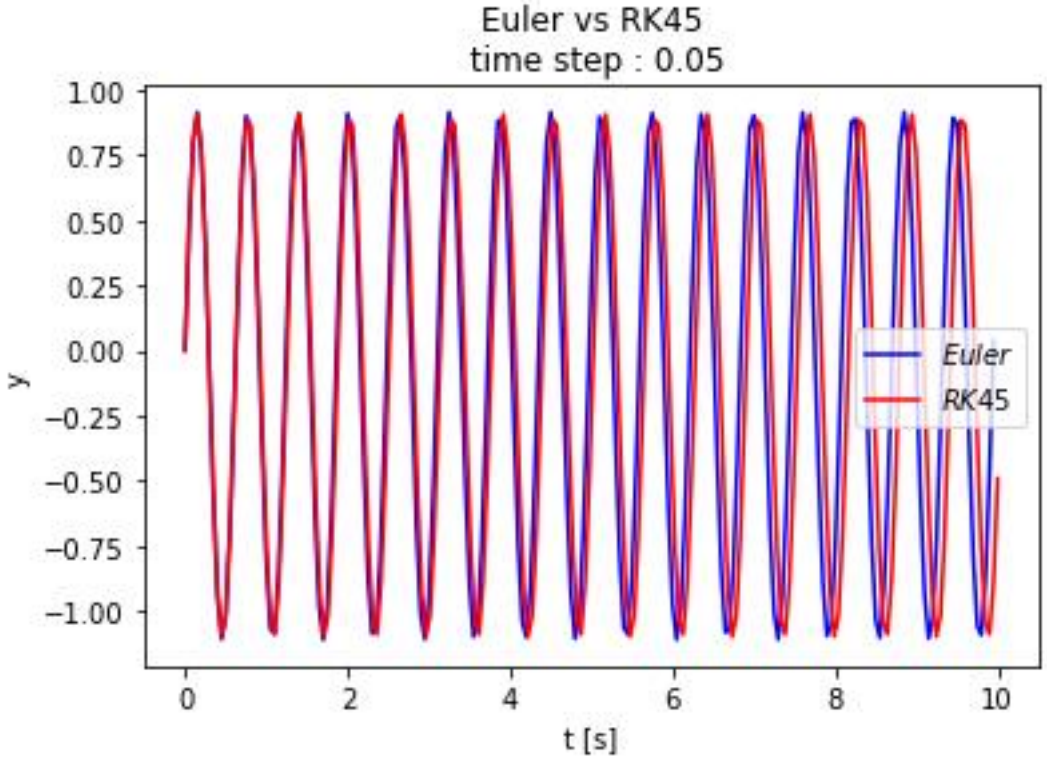
‘LSODA’: Adams/BDF method with automatic stiffness detection. This is a wrapper of the Fortran solver from ODEPACK.

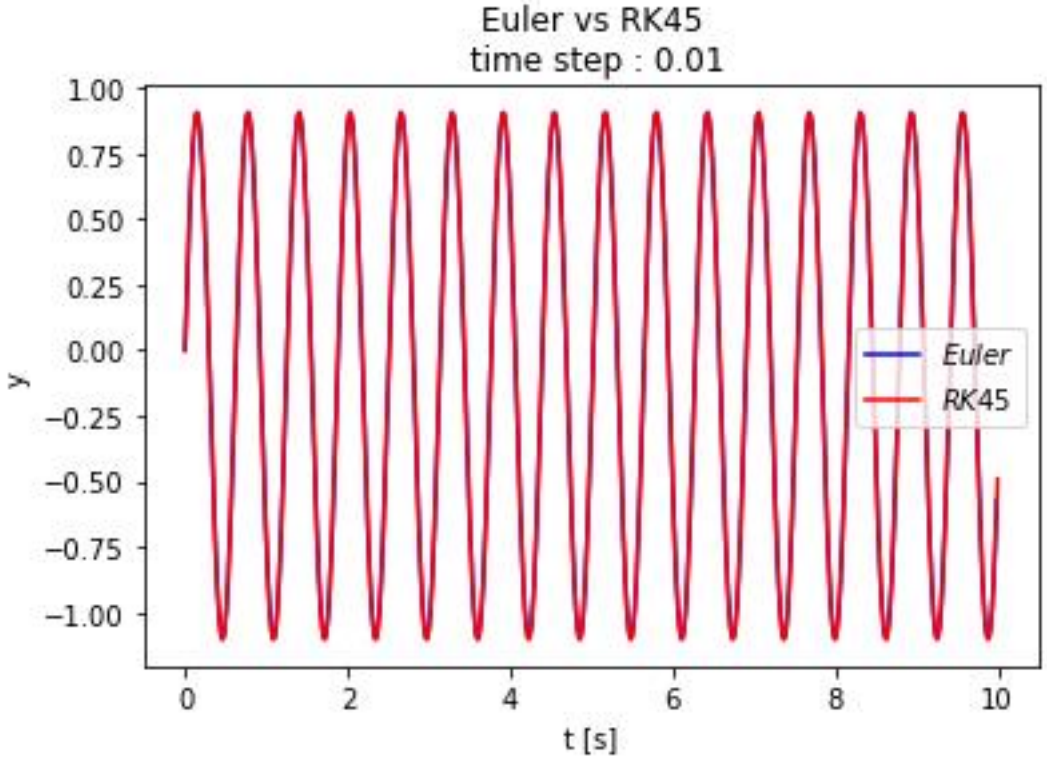
Explicit Runge-Kutta methods ('RK23', 'RK45', 'DOP853') should be used for non-stiff problems and implicit methods ('Radau', 'BDF') for stiff problems. Among Runge-Kutta methods, 'DOP853' is recommended for solving with high precision (low values of rtol and atol).

If not sure, first try to run 'RK45'. If it makes unusually many iterations, diverges, or fails, your problem is likely to be stiff and you should use 'Radau' or 'BDF'. 'LSODA' can also be a good universal choice, but it might be somewhat less convenient to work with as it wraps old Fortran code.

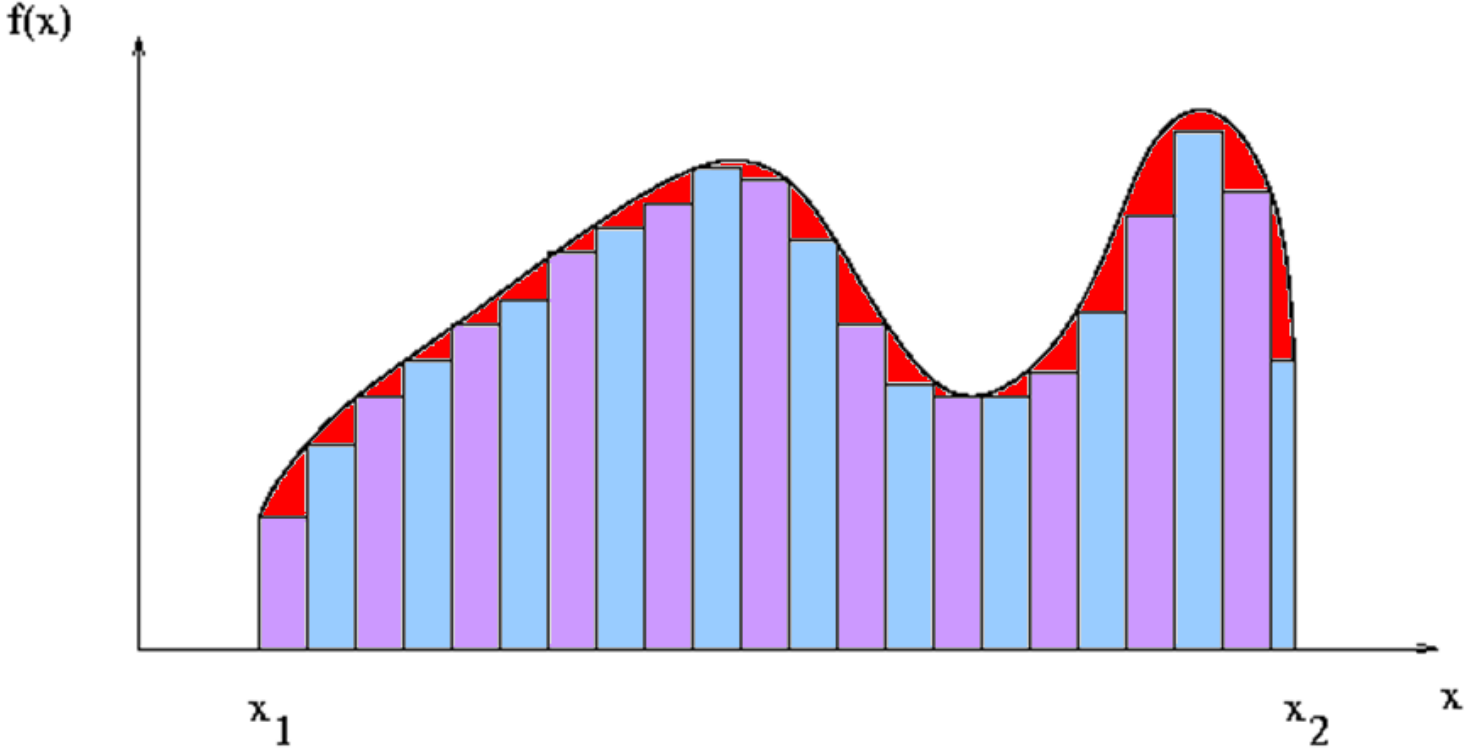

Euler vs RK45
time step : 0.2
Euler
RK45
75
50
25
0
−25
−50
−75
y
0
2
4
6
8
10
t [s]


Euler vs RK45
time step : 0.1
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
y
Euler
RK45
0
2
4
6
8
10
t [s]


Euler vs RK45
time step : 0.05
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
y
0
2
4
6
8
10
t [s]
Euler
RK45


Euler vs RK45
time step : 0.01
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
y
0
2
4
6
8
10
t [s]
Euler
RK45

Numerical Integration



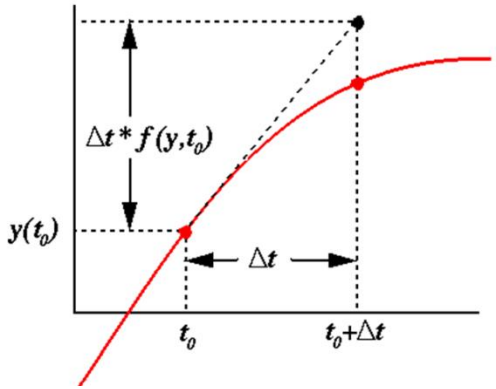
Euler : Euler's method is first order method. It is a straight-forward method that estimates the next point based on the rate of change at the current point. It is a single step method. If no dampening is used, particles get more and more energy over time. "For example, bouncing particles will bounce higher and higher each time". Use this integrator for short simulations or simulations with a lot of dampening where speedy calculations are more important than accuracy.

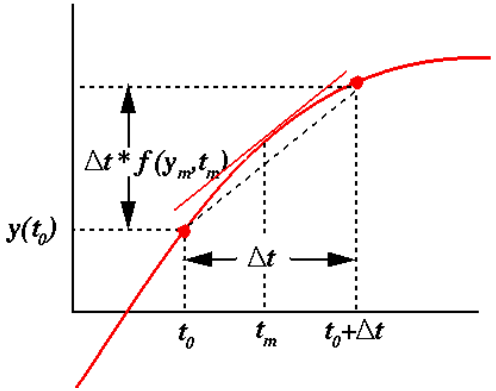
Midpoint : Also known as "**2nd order Runge-Kutta**". Slower than Euler but much more stable.

RK4 : Short for "**4th order Runge-Kutta**". Similar to Midpoint but slower and in most cases more accurate. It is energy conservative even if the acceleration is not constant.

Euler method

Also known as "**Forward Euler**". Simplest integrator. Very fast but also with less exact results. If no damping is used, particles get more and more energy over time. Notably, Forward Euler's method is unconditionally unstable for un-damped oscillating systems (such as a spring-mass system or wave equations) in space discretization.

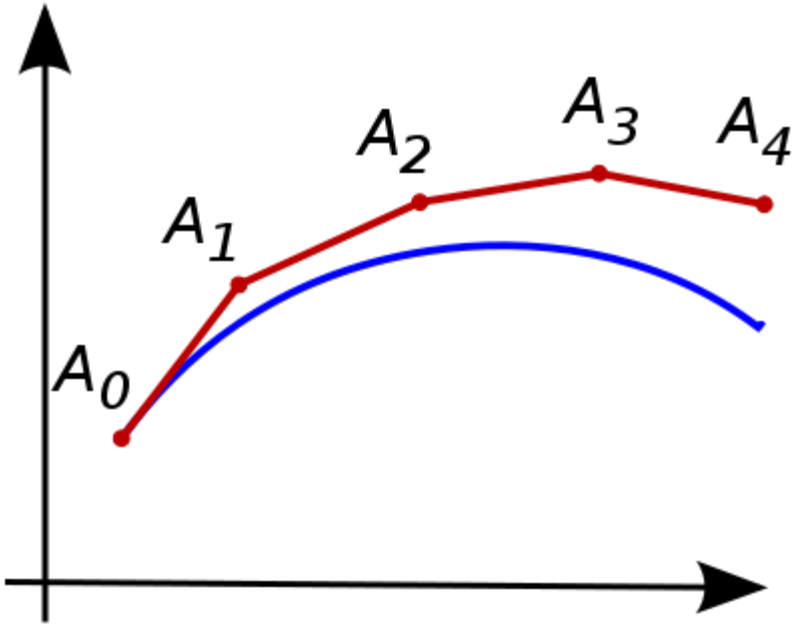


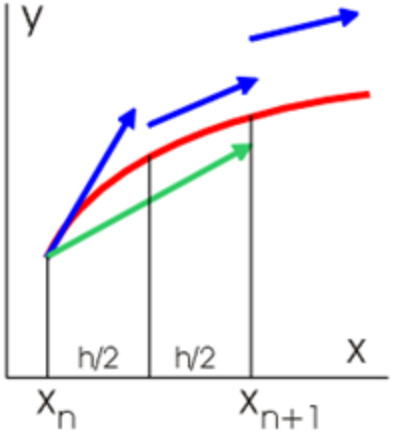


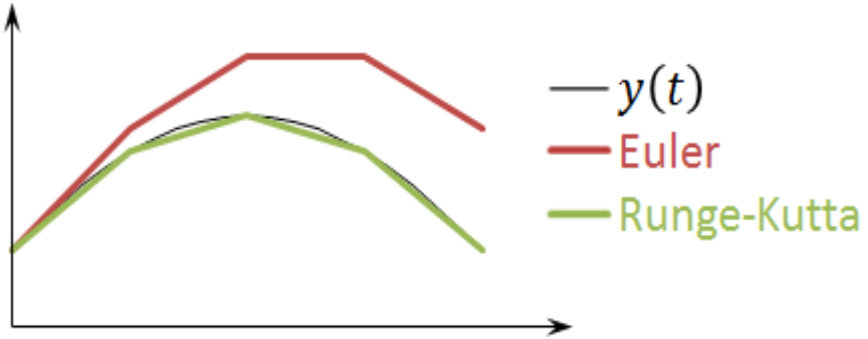
Euler method

Adding physical damping to the model (e.g. Rayleigh damping). In this case the damping will only be applied to the structure and the question is how much damping to introduce when physically it is negligible.

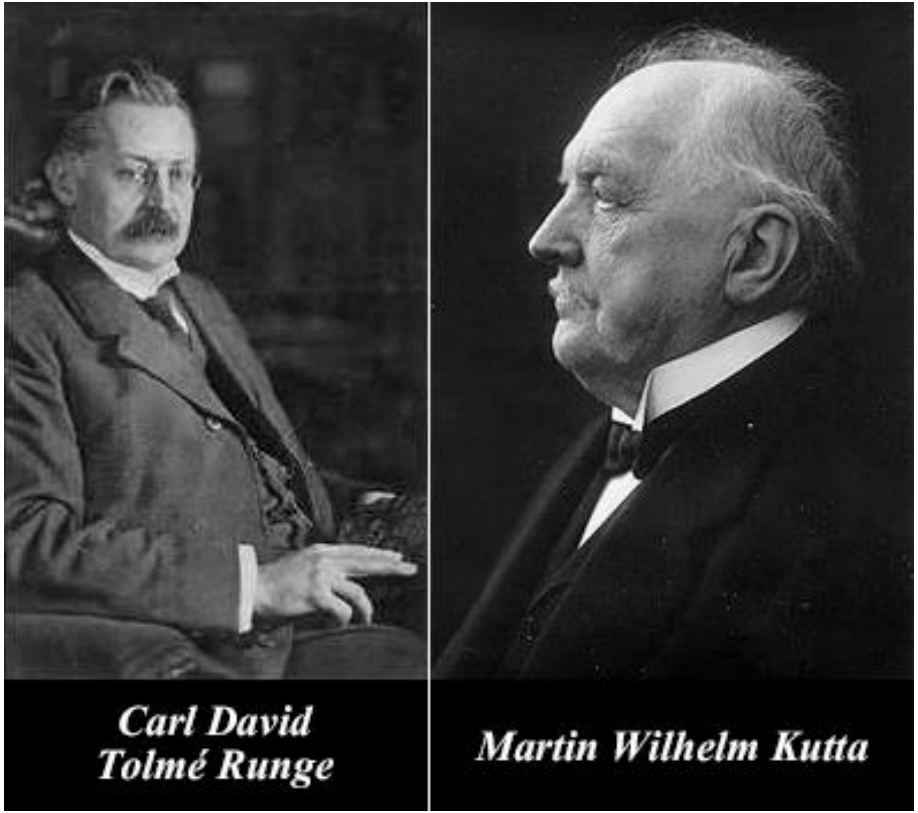
Adding numerical damping. This reduces the numerical oscillations, but also reduces the physical response which should be solved for, and the question is how much numerical damping to introduce in order to obtain acceptable results.


A_0
A_1
A_2
A_3
A_4

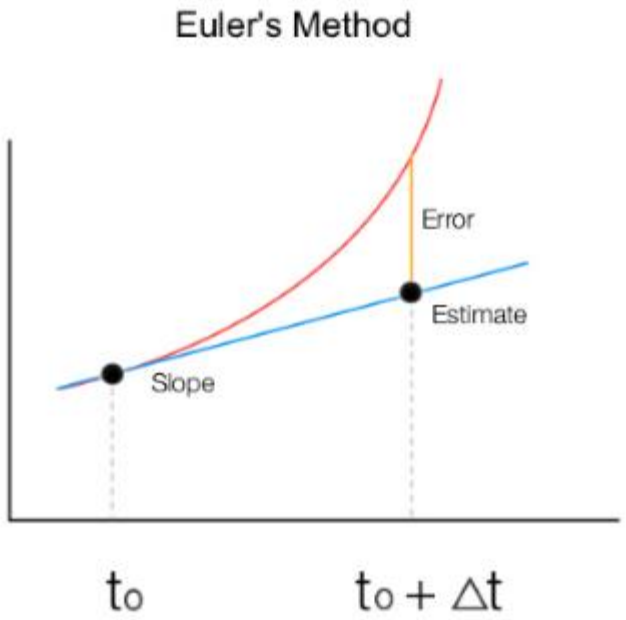

y
h/2
h/2
x
x_n
x_{n+1}


$y(t)$
Euler
Runge-Kutta

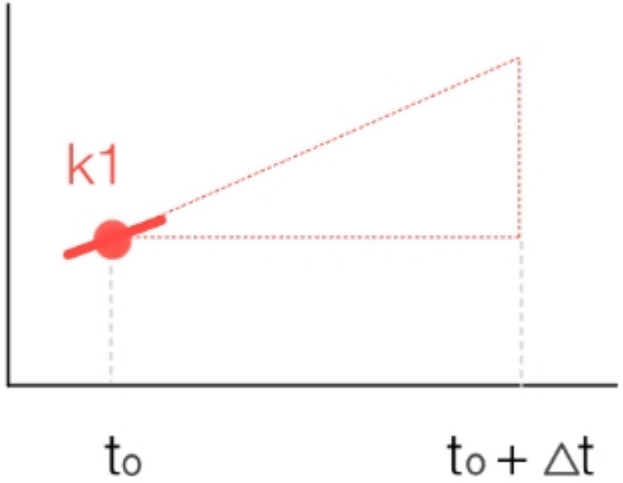
Carl David Tolme Runge – Martin Wilhelm Kutta



Runge –Kutta

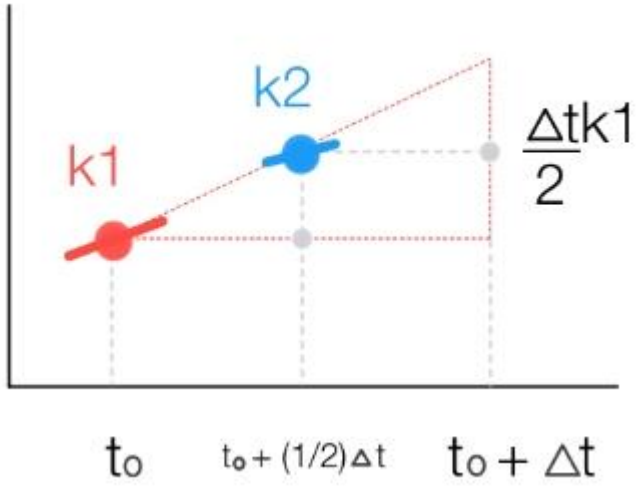


Runge –Kutta



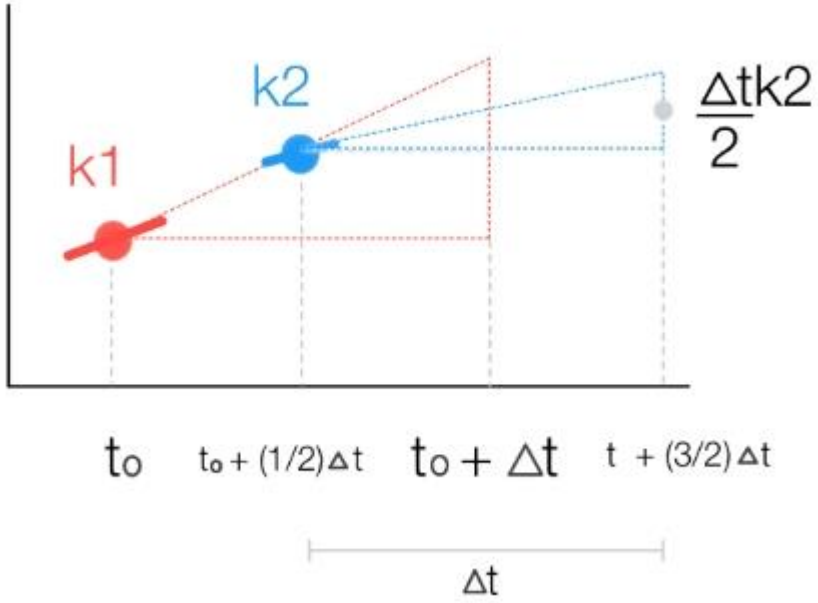
1) At time interval t_0, calculate slope k1.
2) Create a triangle by projecting k1 to $t_0+\Delta t$.

Runge –Kutta



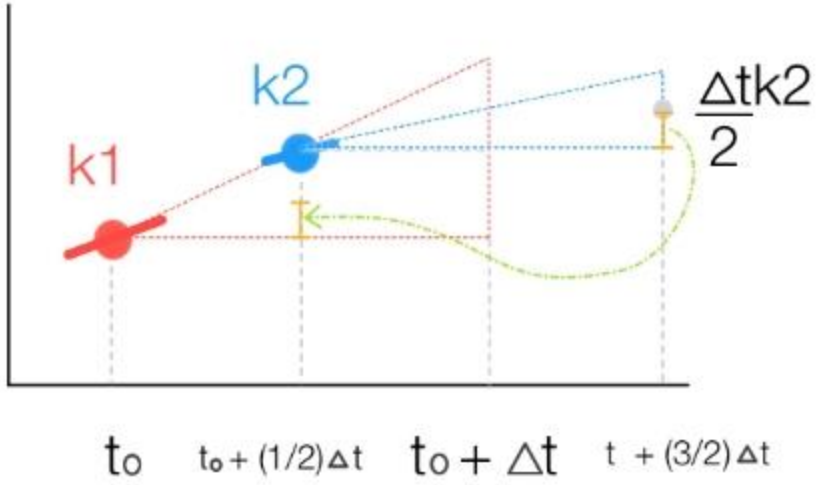
3) Calculate half the height of the triangle, (Δt/2) k1, and draw a horizontal line.
4) Draw a vertical line at interval $t_0+(1/2)\Delta t$.
5) At this intersection point, calculate the slope k2.

Runge –Kutta



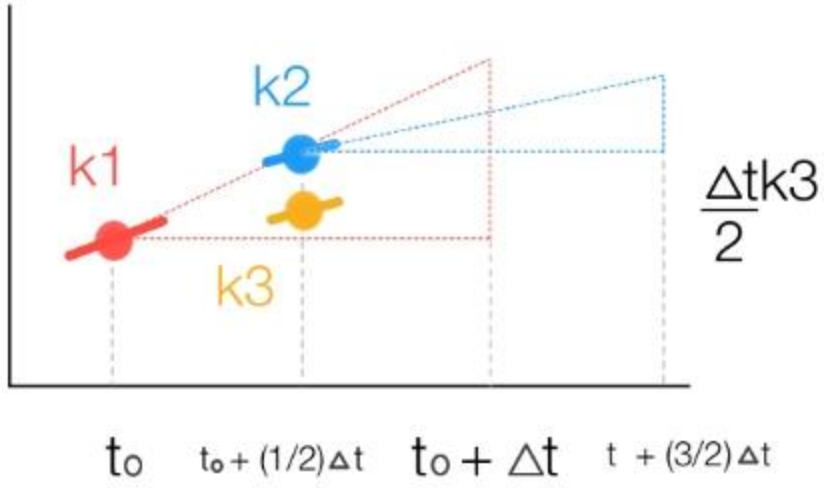
6) Starting out with k2, create a triangle by projecting k2 to t+(3/2)Δt.
7) Calculate half the height of the triangle, (Δt/2)k2.

Runge –Kutta



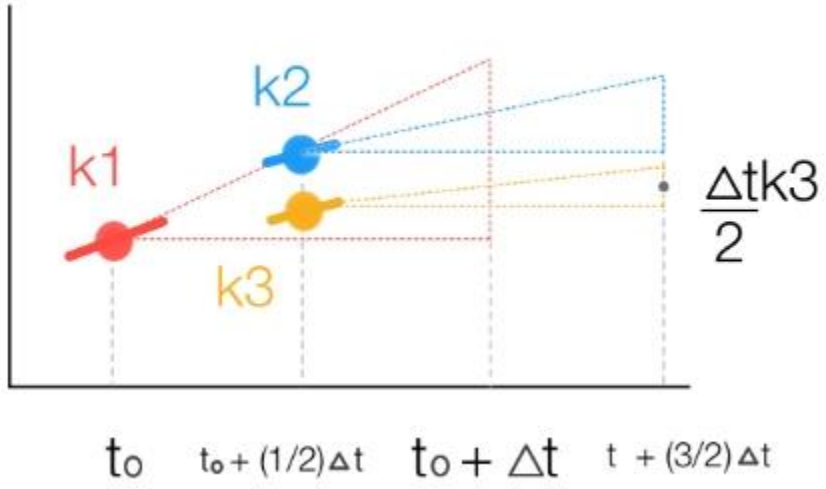
8) Translate the height to the base of the triangle formed by k1 at $t_0+(1/2)\Delta t$.

Runge –Kutta



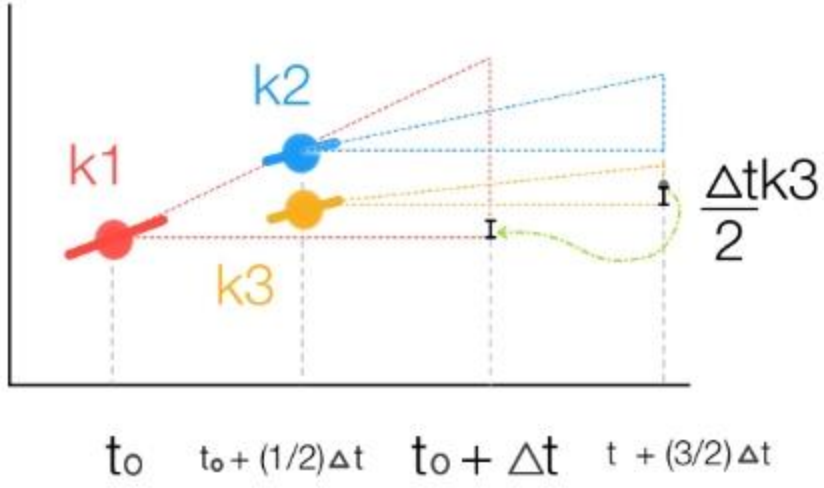
9) At this point, calculate the slope k3.

Runge –Kutta



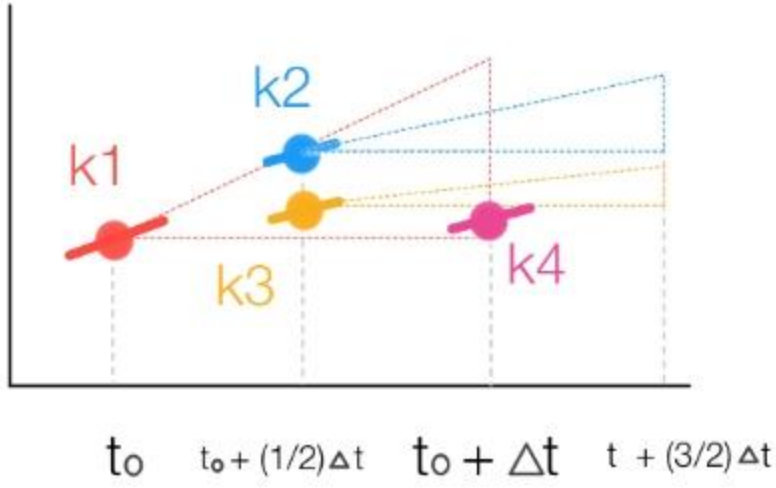
10) Create the another triangle by projecting k3 to $t_0+(3/2)\Delta t$
11) Find half the height of the triangle, $(\Delta t/2)k3$

Runge –Kutta



12) Translate this height to the base of the triangle formed by k1 at point $t_0+\Delta t$

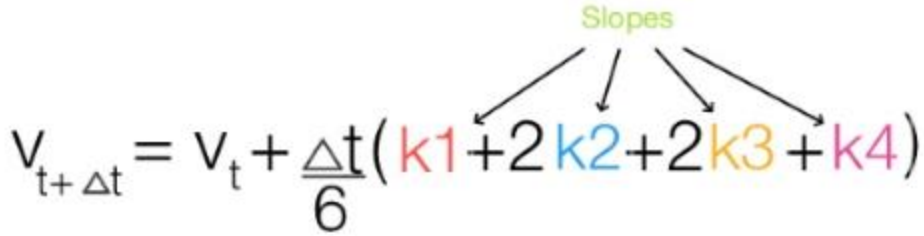
Runge –Kutta



13) Find the slope k4.

Runge –Kutta

The Runga-Kutta uses these slopes as weighted average to better approximate the actual slope, velocity, of the object.



How to set an icon in Pygame.

```
import pygame

WHITE = (255, 255, 255)
pygame.init()
icon = pygame.image.load('icon.png')
pygame.display.set_icon(icon)
size = (700, 200)
screen = pygame.display.set_mode(size)
pygame.display.set_caption("Physics in Game and Animation")
# Loop until the user clicks the close button.
done = False
# Used to manage how fast the screen updates
clock = pygame.time.Clock()
# -------- Main Program Loop -----------
while not done:
    # --- Main event loop
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            done = True

    screen.fill(WHITE)
    pygame.display.flip()
    clock.tick(60)
# Close the window and quit.
pygame.quit()
```