PHYS 2962 – Computing for Physicists

Lecture 7

ORDINARY DIFFERENTIAL EQUATION

Contents for today

- Introduction to ordinary differential equation
- Euler’s method – first-order method
- Python ODE routines
- Excercise

Ordinary Differential Equations

An equation which relates some unknown function x(t) and it's derivatives

$$F\left(x, t, \frac{dx}{dt}, \frac{d^2x}{dt^2}, \frac{d^3x}{dt^3}, \frac{d^4x}{dt^4}, \dots, \frac{d^nx}{dt^n}\right) = 0$$

The highest order derivative (n) which appears in the differential equation determines the order of the differential equation (nth-order ODE).

Examples: $F(x,t) = m\frac{d^2x}{dt^2}$ (Newton's 2nd law, second order ODE)

$\frac{dN}{dt} = -\lambda N$ (Radioactive decay, first order ODE)

Inhomogeneous first-order linear constant coefficient ordinary differential equation

$$\frac{du}{dx} = cu + x^2$$

Homogeneous second-order linear ordinary differential equation

$$\frac{d^2u}{dx^2} - x\frac{du}{dx} + u = 0$$

First-order nonlinear ordinary differential equation

$$\frac{du}{dx} = u^2 + 1$$

Second-order nonlinear ordinary differential equation

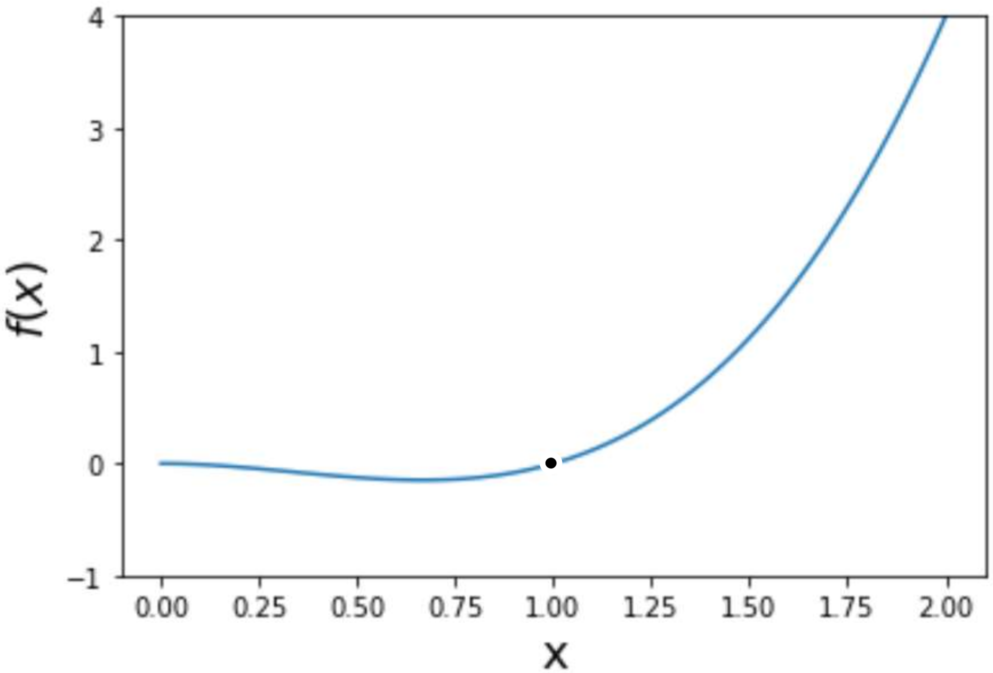
$$L\frac{d^2u}{dx^2} + g\sin u = 0$$

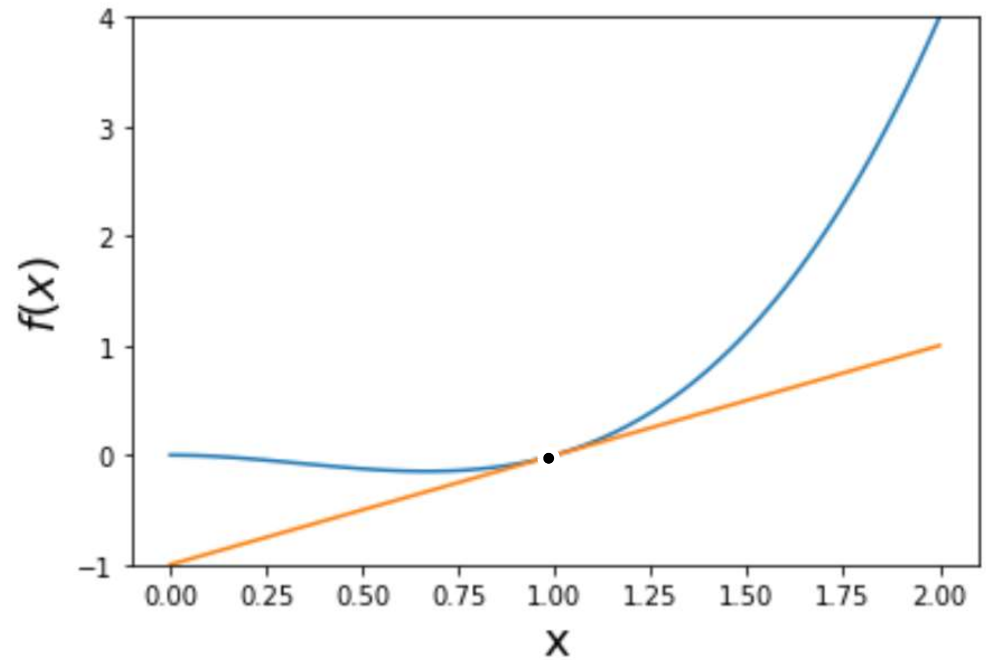
Homogeneous second-order linear constant coefficient ordinary differential equation

$$\frac{d^2u}{dx^2} + \omega^2 u = 0$$

Derivative

$f(x) = x^3 - x^2$





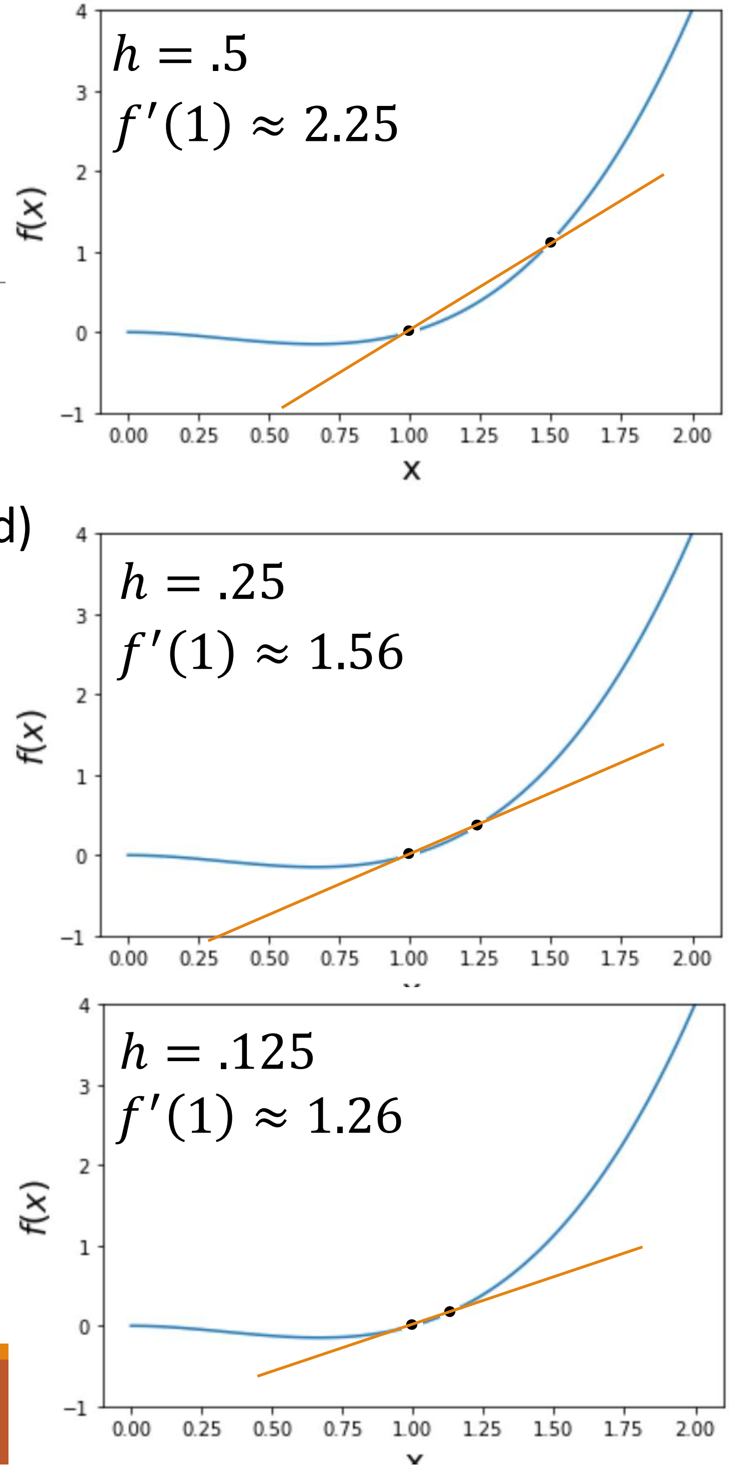
$f'(1)$ is the slope of the line which is tangent at the point $\{1, f(1)\}$.

(analytically, $f' = 3x^2 - 2x$) $\quad f'(1) = 3(1)^2 - 2(1) = 1$

Numeric derivative

$$\frac{dy}{dt} = \lim_{h \to 0} \frac{y(t+h) - y(t)}{h}$$

(Newton's definition: forward difference method)



Only exact in the limit where $h \to 0$.

For any *finite* h, there is some error which depends on the function and the value of h.

In practice you use a very small value of h, but not too small!

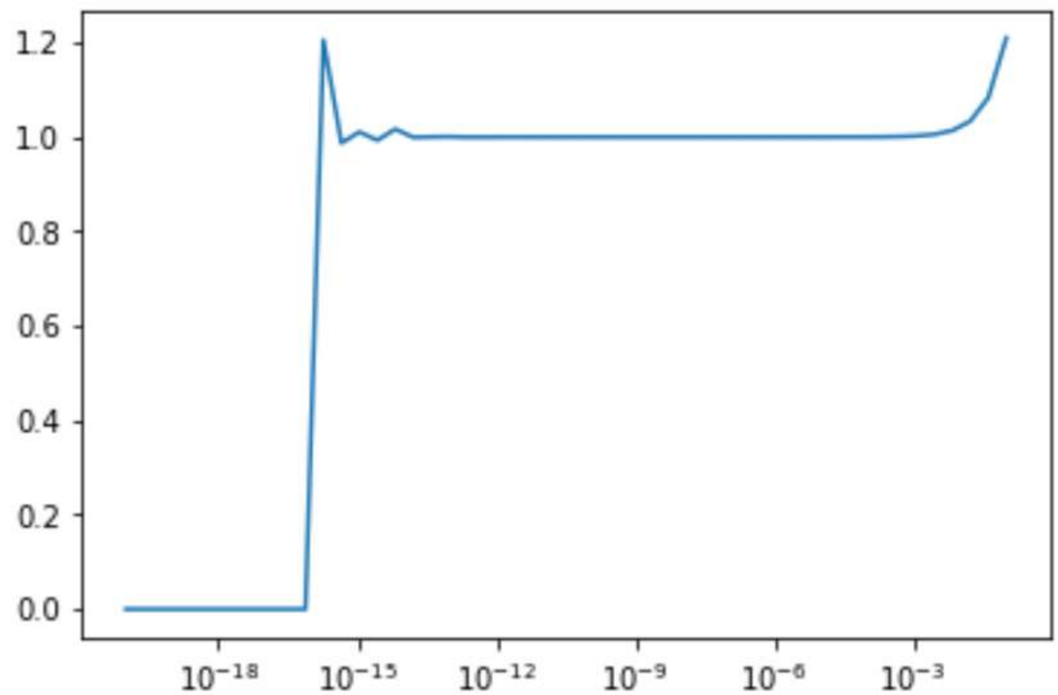
Value of derivative from FD method, for h ranging between 10^{-20} to 10^{-1}

Very small values ($\sim 10^{-16}$) lead to a calculated derivative of 0.

This is because numbers are stored with only finite precision on a computer and $f(1+10^{-16})$ and $f(1)$ are rounded to exactly the same number (round-off error)

For forward difference, the optimal value for $h \sim 10^{-8}$.

```
def fd(f,x,h):
    return (f(x+h)-f(x))/h
def myfun(x):
    return x**3-x**2
h=np.logspace(-20,-1)
ders=fd(myfun,1,h)
plt.semilogx(h,ders)
plt.show()
```



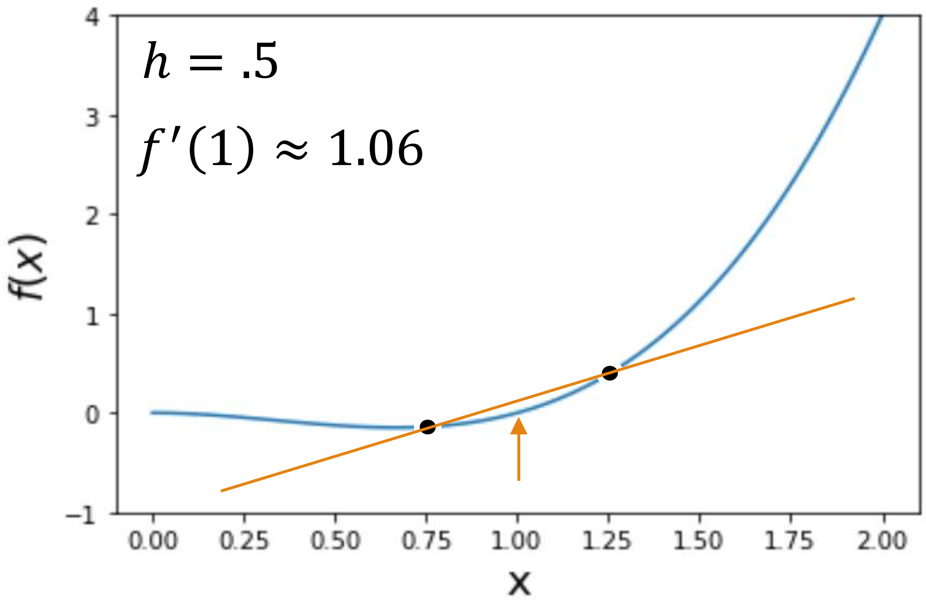
More accurate methods

There are lots of methods to calculate the derivative!

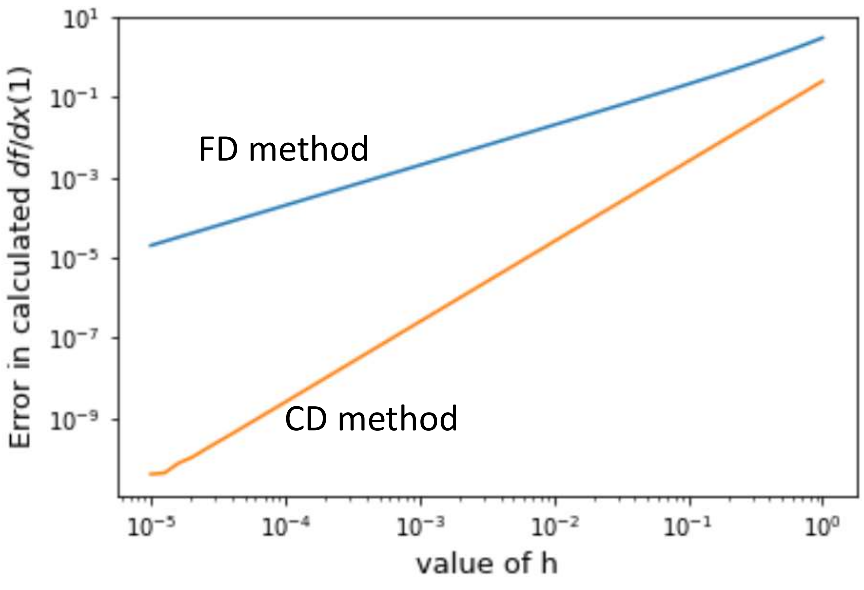
- Forward difference is the worst method (should never be used in practice)

Simplest improvement is the **central difference method (CD)**:

$$\frac{dy}{dt} = \lim_{h \to 0} \frac{y(t + h/2) - y(t - h/2)}{h}$$



FD vs CD



Euler method for solving ODE

$$\frac{dN}{dt} = f(N,t)$$ (first order ODE)

We can use the FD method to rewrite our ODE.

$$\frac{N(t+h) - N(t)}{h} = f(N,t)$$

$$N(t+h) = N(t) + h * f(N,t)$$

If we know $N(t)$, we can determine $N(t+h)$. Then, if we know $N(t+h)$, we can determine $N(t+2h)$, etc. Provided we have an ***initial value,*** we can determine $N(t)$**.**

often written as recurrence relation: $N_{i+1} = N_i + \mathrm{h}f(N_i, ih)$

Example

Radioactive decay: $\frac{dN}{dt} = -.1N$

$$\rightarrow N_{i+1} = N_i + h(-.1N_i)$$

Euler:

$$\frac{dN}{dt} = f(N,t)$$

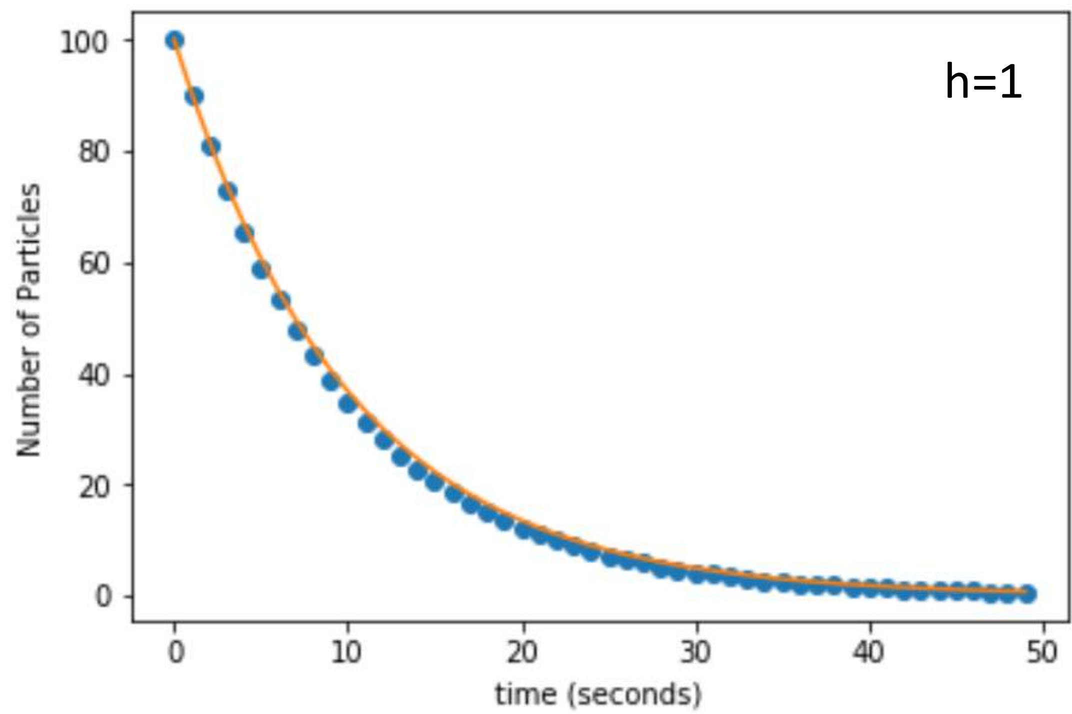
$$N_{i+1} = N_i + \mathrm{h}f(N_i, ih)$$

Lets choose $h = 1$ (time step)

Initial condition: $t_0 = 0, \quad N_0 = 100$

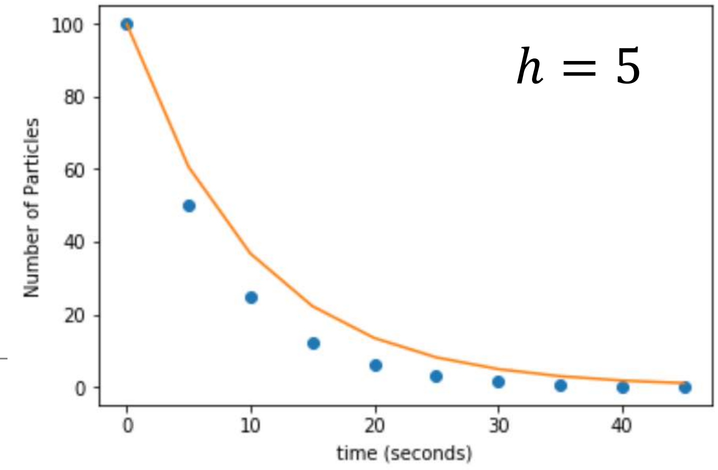
$$
\begin{aligned}
t_0 &= 0, & N_0 &= 100 \\
t_1 &= 1 & N_1 &= 100 + 1(-.1 * 100) = 90 \\
t_2 &= 2 & N_2 &= 90 + 1(-.1 * 90) = 81 \\
t_3 &= 3 & N_3 &= 81 + 1(-.1 * 81) = 72.9
\end{aligned}
$$

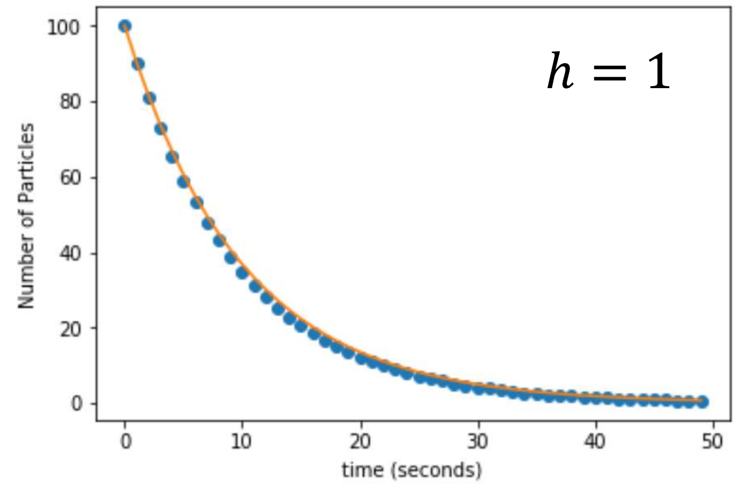
…

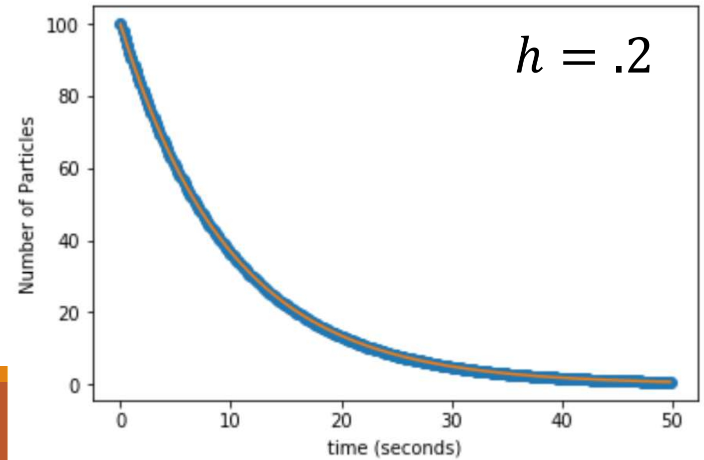


Reducing step size increases accuracy of integration routine.

```
def EUstep(n,h):
    return n+h*(-.1*n)
n=100
t=0
h=.2
Ns=[]
ts=[]
for j in range(int(50/h)):
    ts.append(t)
    Ns.append(n)
    n=EUstep(n,h)
    t=t+h
ts=np.array(ts,dtype=float)
```







Odeint for first-order ODES

scipy.integrate.odeint

scipy.integrate.odeint(*func, y0, t, args=(), Dfun=None, col_deriv=0, full_output=0, ml=None, mu=None, rtol=None, atol=None, tcrit=None, h0=0.0, hmax=0.0, hmin=0.0, ixpr=0, mxstep=0, mxhnil=0, mxordn=12, mxords=5, printmessg=0, tfirst=False*) [source]

Integrate a system of ordinary differential equations.

Solves the initial value problem for stiff or non-stiff systems of first order ode-s:

```
dy/dt = func(y, t, ...)  [or func(t, y, ...)]
```

where y can be a vector.

For 1st order ODEs. The only variables in the equation can be (y,y',t).

Need to make a python function which accepts (y,t) and returns dy/dt

$$\frac{dN}{dt} = -.1N$$

This is a 1st order ODE dealing with $t, N, dN/dt$. We want to know the unknown function $N(t)$.

Need to construct a python function which accepts N, t and returns $\frac{dN}{dt}$.

```
from scipy.integrate import odeint

#This function defines the ODE
# I'm solving. (radioactive decay)
def f(y,t):
    dydt=-0.1*y
    return dydt
```

our ODE does not explicitly depend on t. That's fine, if we are given (N,t) we known what dN/dt is.

once our ODE is defined,
it can be solved easily:

```
times=np.linspace(0,50,100)
intvalue=100
N=odeint(f,intvalue,times)
```

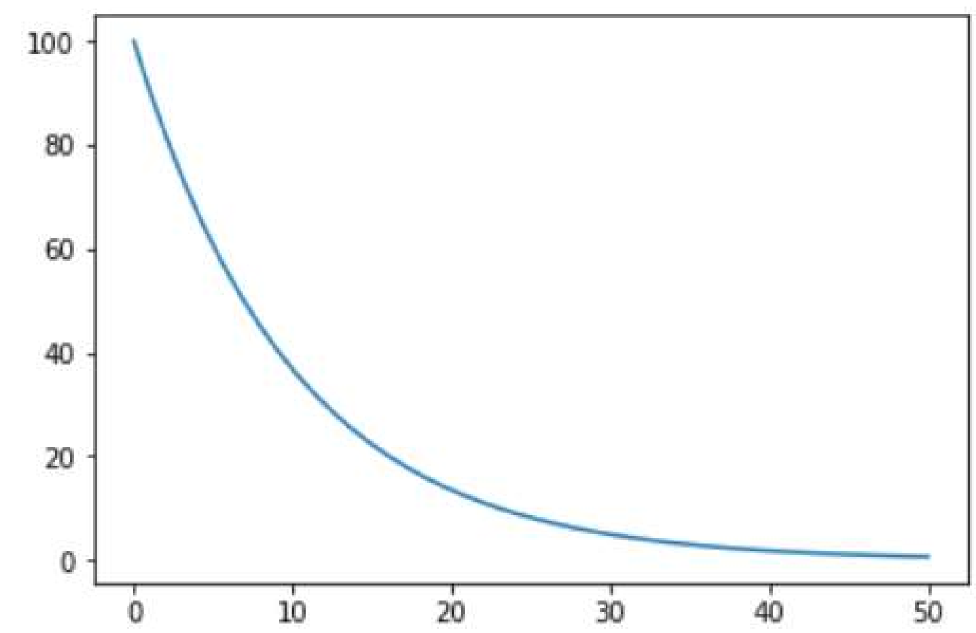
odeint returns array of solved values of function for given array of times.

need to provide initial value.

```
from scipy.integrate import odeint

# This function declaration defines
# to ODE I'm solving.
def f(y,t):
    dydt=-.1*y
    return dydt

times=np.linspace(0,50,100)
intvalue=100
N=odeint(f,intvalue,times)
plt.plot(times,N)
plt.show()
```



General Comments

$$\frac{dy}{dt} = g^3(t)y(t) \qquad \text{(linear)}$$
$$\frac{dy}{dt} = \lambda y(t) - \lambda^2 y^2(t) \qquad \text{(nonlinear)}$$

- The general solution of a first-order differential equation always contains one arbitrary constant.
- A general solution of a second-order differential equation contains two such constants, and so forth.

- For any specific problem, these constants are fixed by the initial conditions.

- Regardless of how powerful a computer you use, the mathematical fact remains and you must know the initial conditions in order to solve the problem.

Reduction to a system of 1st order ODEs

The Objective is to transform our equation into:

$$\begin{aligned}
\frac{dy^{(0)}}{dt} &= f^{(0)}(t, y^{(i)}) \\
\frac{dy^{(1)}}{dt} &= f^{(1)}(t, y^{(i)}) \\
&\ddots \\
\frac{dy^{(N-1)}}{dt} &= f^{(N-1)}(t, y^{(i)})
\end{aligned}$$

We can reduce an n^{th} order ODE to n 1^{st} order ODEs

Converting an order-*n* ODE to *n* first-order ODE's

Newton's second law: $F = ma$ $\quad F = m\dfrac{d^2x}{dt^2}$ $\quad F = F\left(t, x, \dfrac{dx}{dt}\right)$

second-order ODE
$$m\frac{d^2x}{dt^2} = F\left(t, x, \frac{dx}{dt}\right)$$

How to convert to two coupled 1st order ODEs?

Most ODE solvers require the ODE to be defined in terms of a set of 1st order ODEs

Converting an order-*n* ODE to *n* first-order ODE's

Newton's second law: $F = ma$ $\quad F = m\frac{d^2x}{dt^2}$ $\quad F = F\left(t, x, \frac{dx}{dt}\right)$

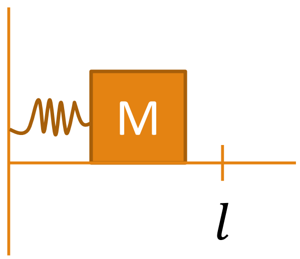
second-order ODE $\quad m\frac{d^2x}{dt^2} = F\left(t, x, \frac{dx}{dt}\right)$

$$\begin{cases} \frac{dx}{dt} = v(t) \\ \frac{dv}{dt} = \frac{F(t, x, v)}{m} \end{cases}$$

Most ODE solvers require the ODE to be defined in terms of a set of 1st order ODEs

2nd order ODE → two coupled 1st order ODEs

Mass on a spring



Hooke's Law
$F = -k(x - l)$

$$\mathrm{M}\frac{d^2x}{dt^2} = -k(x - l)$$ (second order ODE)

1^{st} step: reduce it to two 1^{st} order ODEs

$$\frac{dx}{dt} = v$$

$$\frac{dv}{dt} = -\frac{k(x-l)}{M}$$

(nice to arrange things so derivatives are isolated on the left. Easier to convert to code)

Odeint can handle sets of 1^{st} order equations. In this case y is an array of the functions to be solved for in the problem, i.e. $y = [x, v]$

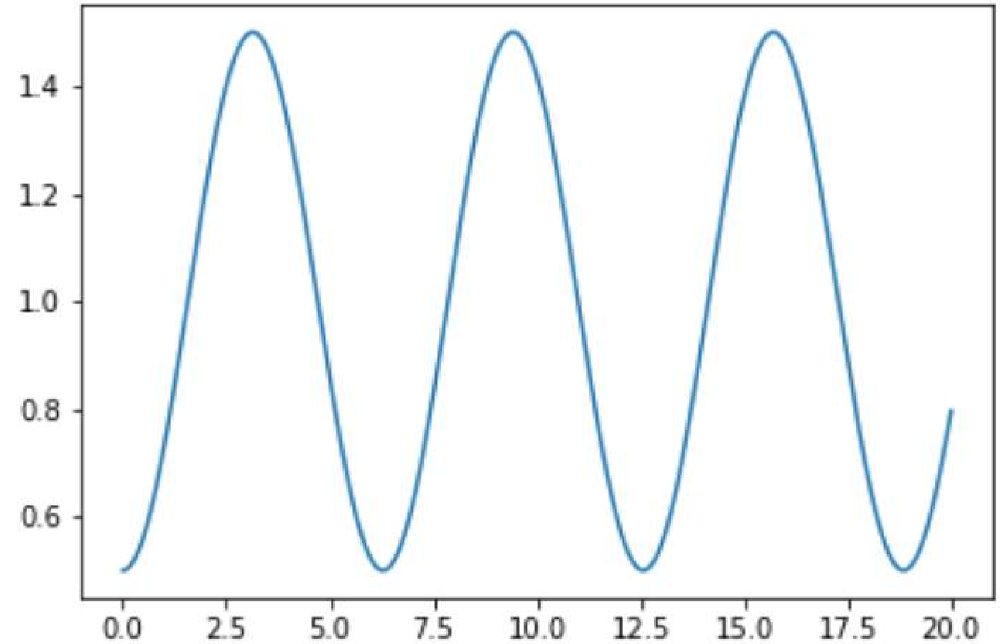
To define our set of ODEs: **we need to define a function which accepts variables {[x,v],t} and returns** $[\boldsymbol{\frac{dx}{dt}}, \boldsymbol{\frac{dv}{dt}}]$.

$$\frac{dx}{dt} = v$$

$$\frac{dv}{dt} = -\frac{k(x-l)}{M}$$

```
def f(y,t):
    l=1;k=1;M=1;
    x,v=y
    dxdt=v
    dvdt=-k*(x-l)/M
    return [dxdt,dvdt]
```

```
times=np.linspace(0,20,1000)
sol=odeint(f,[.5,0],times)
plt.plot(times,sol[:,0])
plt.show()
```



returns both unknown functions x(t) and v(t) to sol.

Euler vs. odeint