Numerical solution to 1st order and 2nd order Ordinary Differential Equation (ODE)

Dr. Shyamal Bhar
Assistant Professor
Department of Physics
Vidyasagar College for Women

In this section we shall discuss different numerical methods to solve Ordinary Differential Equation (ODE) for both first order and second order. ODEs are mostly used in many branches in Physics. For example dynamical systems in Physics are described by a second order ODE $m\frac{d^2x}{dt^2} = F(x, \frac{dx}{dt}, t)$. There are many equations in Physics which is governed by second order ODE eg Simple Harmonics motion, Damped and Forced vibrations, LC, LCR circuits etc. Now any second order ODE can be decomposed into two first order coupled ODEs. For example above equation can be written as $\frac{dx}{dt} = v$ and $m\frac{dv}{dt} = F(x,v,t)$. That's why here we shall learn different numerical techniques to solve first order ODE.

The general form of first order ODE is

$$\frac{dy}{dx} = f(x, y) \qquad \text{.....(1)}$$

with the initial condition $y = y_0$ at $x = x_0$. Here y is the dependent variable and x is the independent variable. The problem is basically an Initial value problem.

To solve this equation at a point x, we divide the range $[x_0, x]$ into a large number of tiny intervals of equal width (h). For n number of intervals $h = x_{n+1} - x_n$.

The general form of a second order ODE is

$$\frac{d^2y}{dx^2} + \lambda\frac{dy}{dx} + ky = F$$

This second order ODE can be split into two first order coupled equation as

$$\frac{dy}{dx} = z = f_1(x, y, z)$$

$$\& \quad \frac{dz}{dx} = (F - ky - \lambda z) = f_2(x, y, z)$$

Solving these two ODEs simultaneously we can solve for y.

Now the equation (1) can be written in a integral form as

$$y_{n+1} = y_n + \int_{x_n}^{x_{n+1}} f(x, y)\, dx \qquad .....(2)$$

Where y_n is the solution (y) of the differential equation at $x = x_n$. The integration of the equation (2) can be evaluated numerically by using different approximation methods. Depending on the approximations used, there are four techniques to solve ODEs. These are i) Euler Method ii) Modified Euler method iii) Runge-Kutta second order (RK2) method and iv) Runge-Kutta fourth order (RK4) method.

1. Euler method:

In the first approximation we can evaluate the integration of equation (2) by assuming the function then we get $f(x, y) = f(x_n, y_n)$

$$y_{n+1} = y_n + h\, f(x_n, y_n) \qquad .....(3)$$

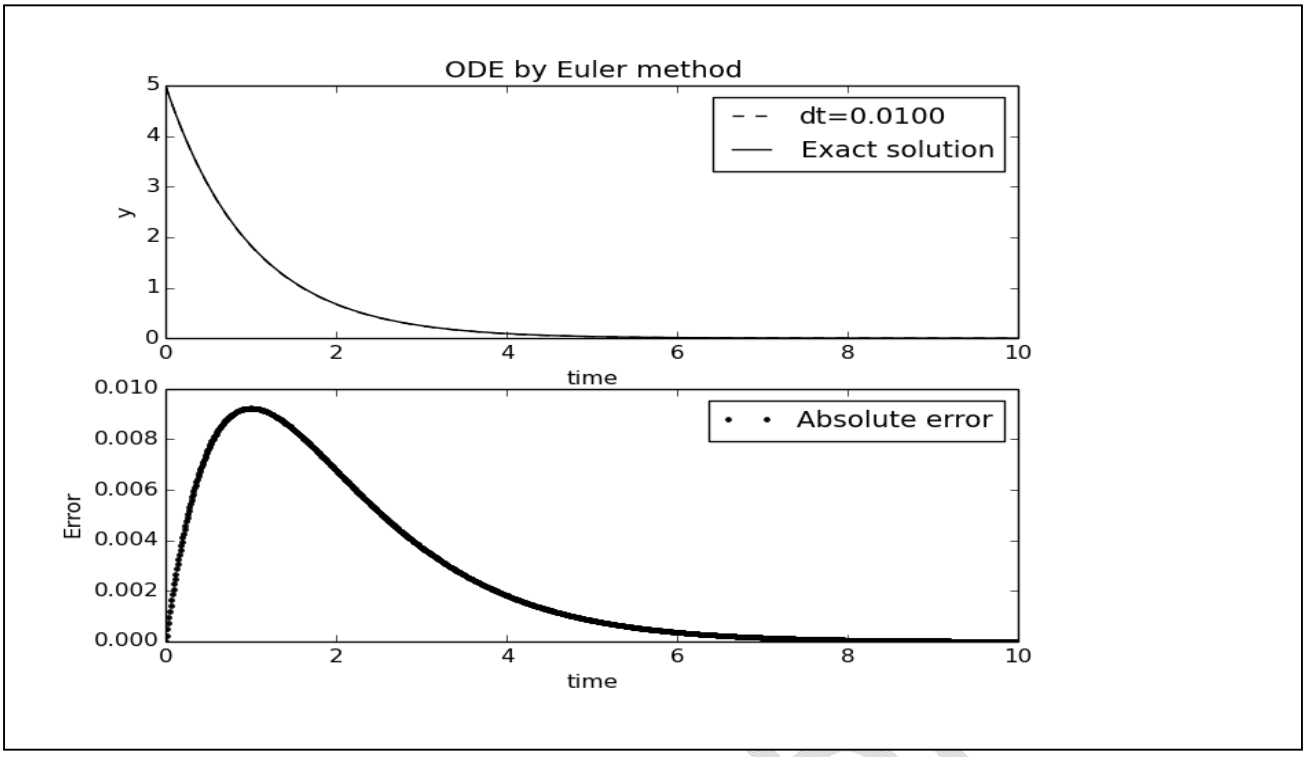
Above equation is the update rule of **Euler method or Euler forward method**. The iteration starts from some initial value $y(x_0) = y_0$. So primary requirement for Euler method is, the problem must be ***initial value problem***. We find solution for certain range of x, and the range is divided into tiny intervals.

Example 1:

Radioactive decay is governed by the equation $\frac{dy}{dt} = -ky$. This is an exponential decay equation and the exact solution is given by $y = y_0 e^{-kt}$, where y_0 is the initial value of y at $t = 0$. We shall solve this equation by Euler method.

```
# Solution of Radio-active decay equation (1st order ODE)
# Euler method
import matplotlib.pyplot as plt
import numpy as np
```

```
# initial value of t, y
k,t,y=1,0.0,5.0
y0=y
tt,yy=[],[]
tf=10 # final value of x

dt=0.01 # step length

# defining function

def f(t,y,k):

      return -k*y

# implementation of Euler method
while t<=tf:
      tt.append(t)
      yy.append(y)
      y=y+dt*f(t,y,k)
      t+=dt

#calculation of exact result
tt=np.array(tt)
exact=y0*np.exp(-k*tt)

# Plotting solution with exact result
plt.subplot(2,1,1)
plt.plot(tt,yy,'k--',label="dt=%.4f"%(dt))
plt.plot(tt, y0*np.exp(-k*tt),'k',label="Exact solution")
plt.xlabel("time")
plt.ylabel("y")
plt.legend(loc='best')

# Plotting absolute error
diff=exact-yy
plt.subplot(2,1,2)
plt.plot(tt,diff,'k.',label="Absolute error")
plt.xlabel("time")
plt.ylabel("Error")
plt.legend(loc='best')
plt.title("ODE by Euler method")
plt.savefig("decay-euler.png")
```



Example 2:

Growth of current in series LR circuit. Supplose the LR circuit is excited by a battery of emf E.

The differential equation for the growth of current in LR circuit is

$$L\frac{di}{dt} + Ri = E$$

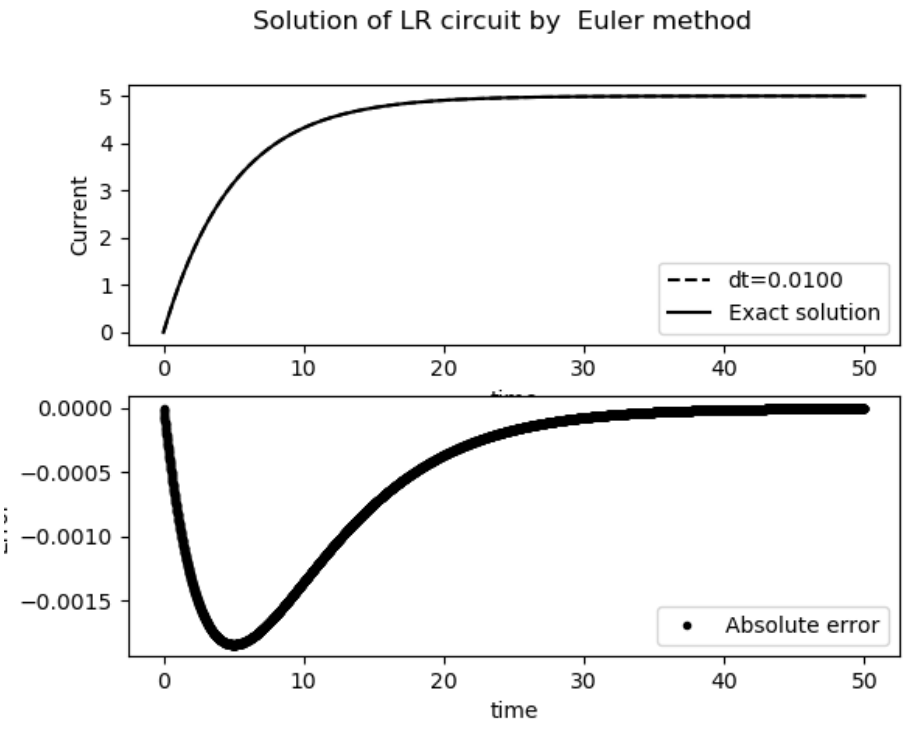
$$\Rightarrow \frac{di}{dt} = (E - Ri)/L$$

$$\therefore f(i,t) = \frac{(E - Ri)}{L}$$

Python code for Euler method and result for the above differential equation is

```
# Solution of LR circuit (1st order ODE): Growth of current
#  Euler method
import matplotlib.pyplot as plt
import numpy as np

# initial value of t, i
e=1.0
r=0.2
l=1.0

t=0.0
i=0.0

tt=[]
ii=[]
tf=50.0 # final value of t
dt=0.01 # step length

# defining function
def f(t,i):
      return (e-i*r)/l
```

```
while t<=tf:
      tt.append(t)
      ii.append(i)
      i+=dt*f(t,i)
      t=t+dt

# Exact solution
tt=np.array((tt))
ii=np.array((ii))
exact=(e/r)*(1.0-np.exp(-r/l*tt))
# Plotting of solution with exact result
plt.subplot(2,1,1)
plt.plot(tt,ii,'k--',label="dt=%.4f"%(dt))
plt.plot(tt, exact,'k',label="Exact solution")
plt.xlabel("time")
plt.ylabel("Current")
plt.legend(loc='best')

# Calculation of Error and plotting
diff=exact-ii

plt.subplot(2,1,2)
plt.plot(tt,diff,'k.',label="Absolute error")
plt.xlabel("time")
plt.ylabel("Error")
plt.legend(loc='best')
plt.suptitle("Solution of LR circuit by  Euler method")
plt.savefig("LR-euler-error.png")
```



Example 3:

Charging of capacitor in series RLC circuit. The emf equation for the circuit is

$$L\frac{d^2q}{dt^2}+R\frac{dq}{dt}+\frac{q}{C}=E$$

Where $q(t)$ is the instantaneous charge on the capacitor. This second order ODE can be split into two first order coupled ODEs,

$$\frac{dq}{dt}=i=f_1(q,i,t)$$
$$\frac{di}{dt}=\left(E-Ri-\frac{q}{c}\right)/L=f_2(q,i,t)$$

There are three cases of interest i) Overdamped when $R>2\sqrt{\frac{L}{C}}$

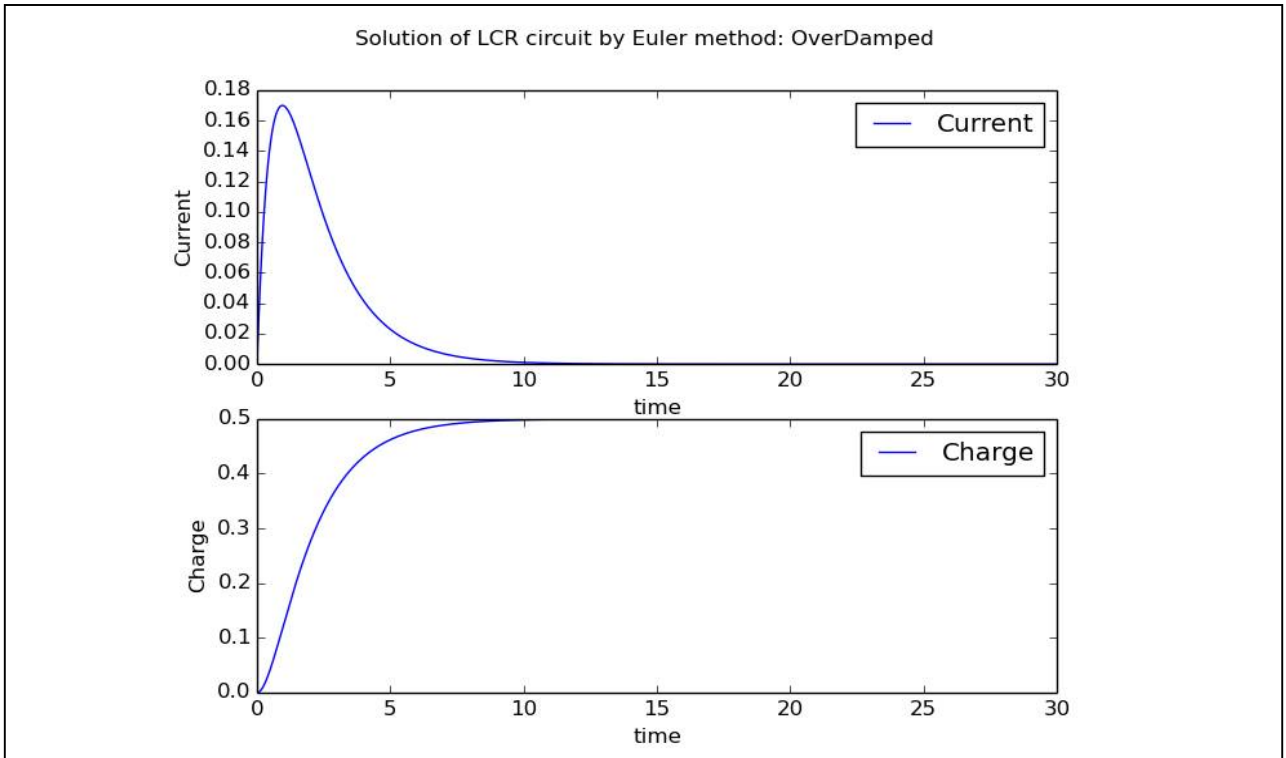
ii) Critical damped when $R=2\sqrt{\frac{L}{C}}$

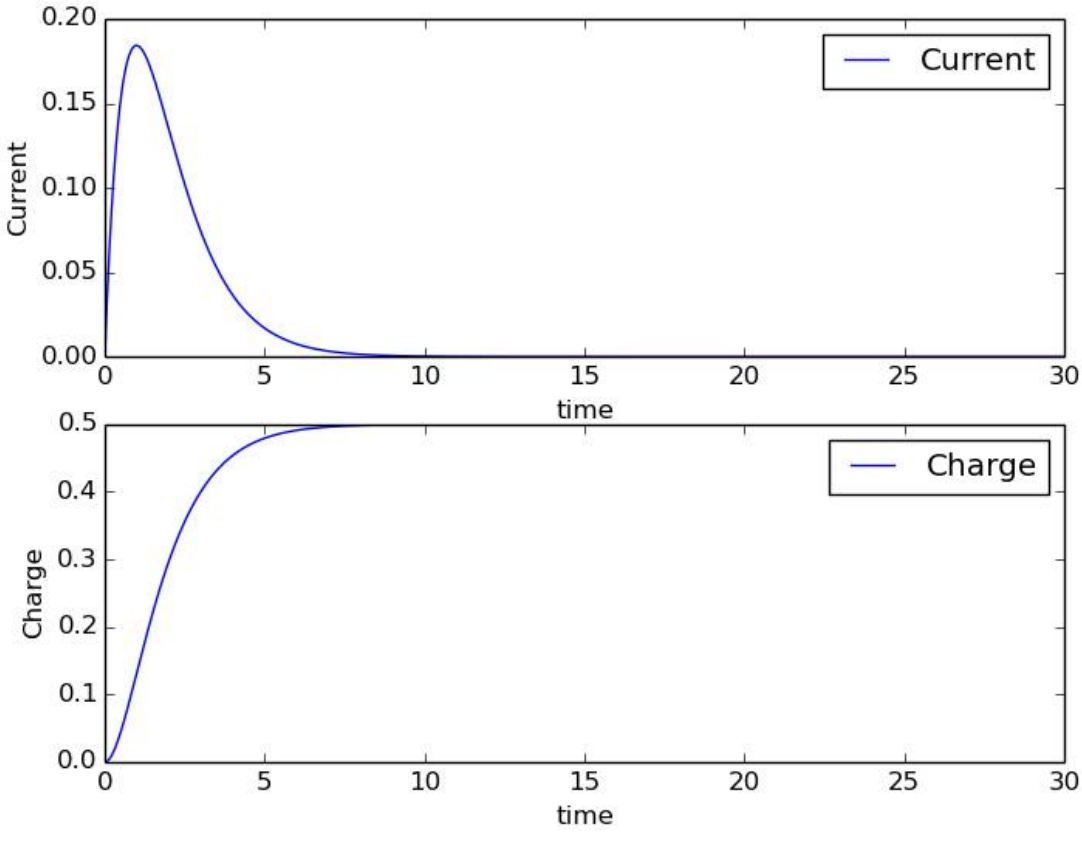
and iii) Underdamped when $R<2\sqrt{\frac{L}{C}}$

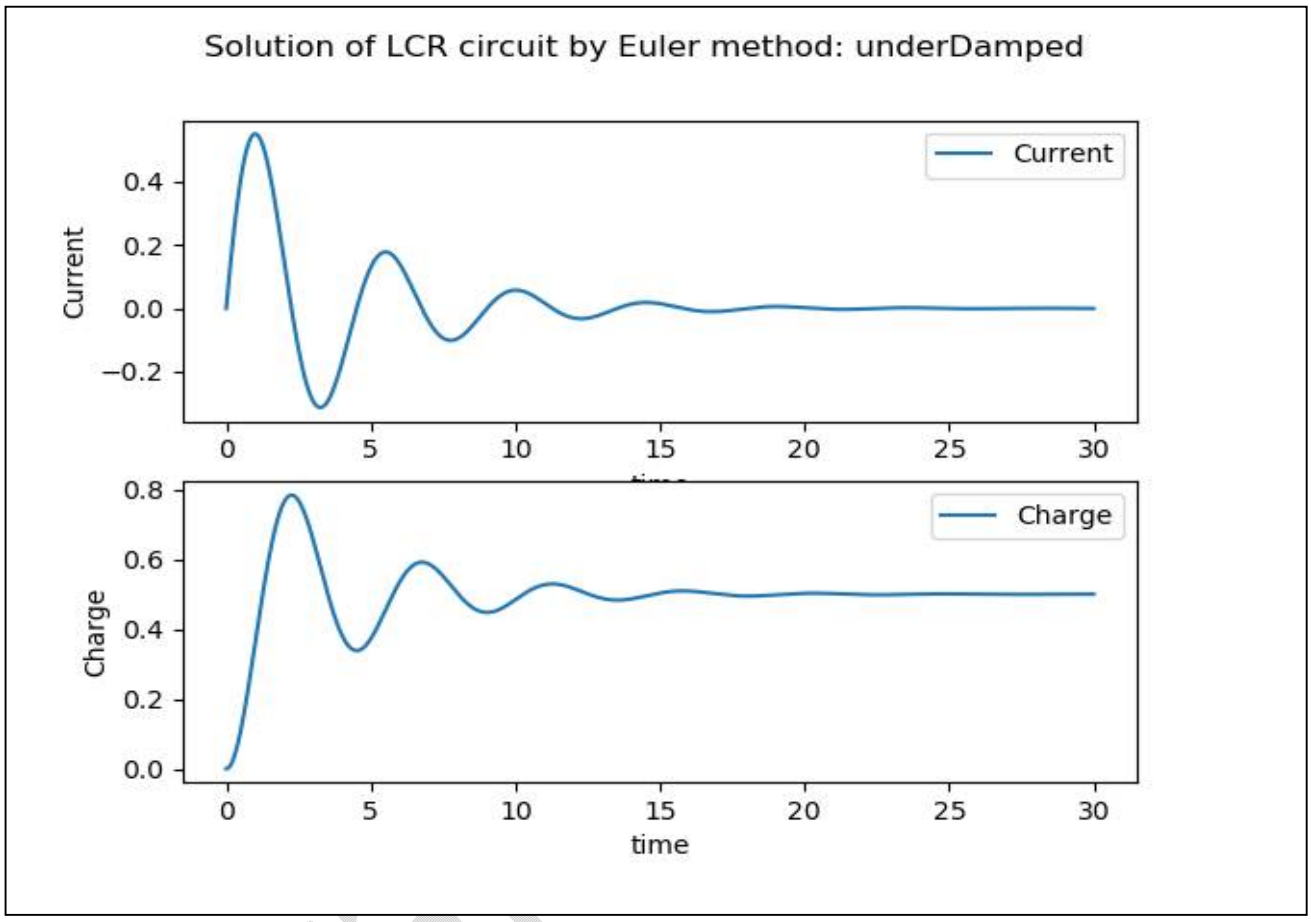
Python code for Euler method for solving above equation. You need to change the values of L, C and R for studying different cases.

```
# Solution of LCR circuit (2nd order ODE): Charging of Capacitor
# Euler method
import matplotlib.pyplot as plt
import numpy as np
#"""
# initial value of t, i
##########################################
###### Case-1, Over damped case R>2*sqrt(L/C)
e=1.0
l=2.0
c=0.5
r=4.5
##########################################
#"""
"""
##########################################
###### Case-2, Critical damped case R=2*sqrt(L/C)
e=1.0
l=2.0
c=0.5
r=4
##########################################
"""
```

```
"""
##############################################
###### Case-3, Underdamped case R<2*sqrt(L/C)
e=1.0
l=1.0
c=0.5
r=0.5
##############################################
"""
t=0.0
i=0.0
q=0.0
tf=30.0 # final value of t
dt=0.01 # step length

tt=[]
ii=[]
qq=[]

# defining function
def f1(q,i,t):
        return i #  dq/dt=i ..> f1(q,i,t)

def f2(q,i,t):
        return (e-i*r-q/c)/l  # di/dt=(e-ri-q/c)/l ..> f2(q,i,t)

while t<=tf:
        tt.append(t)
        qq.append(q)
        ii.append(i)
        q+=dt*f1(q,i,t)
        i+=dt*f2(q,i,t)
        t+=dt
plt.subplot(2,1,1)
plt.plot(tt,ii,label="Current")
plt.xlabel("time")
plt.ylabel("Current")
plt.legend(loc='best')

plt.subplot(2,1,2)
plt.plot(tt,qq,label="Charge")
plt.xlabel("time")
plt.ylabel("Charge")
plt.legend(loc='best')
plt.suptitle("Solution of LCR circuit by Euler method: OverDamped")
plt.savefig("Overdamped.png")
plt.show()
```


Solution of LCR circuit by Euler method: OverDamped
Current
Charge
time
time


Solution of LCR circuit by Euler method: Critical Damped
Current
Charge
time
time


Solution of LCR circuit by Euler method: underDamped
Current
Charge
time

2. Modified Euler method:

If we approximate the integration of equation (2) by Trapezoidal method , then the equation takes the form

$$y_{n+1} = y_n + \frac{h}{2}\left[f\left(x_n, y_n\right) + f\left(x_{n+1}, y_{n+1}\right)\right]$$

Use Euler method to Approximate $y_{n+1} \approx y_n + h\, f(x_n, y_n)$ in the above equation we get

$$y_{n+1} = y_n + \frac{h}{2}\left[f\left(x_n, y_n\right) + f\left(x_{n+1},\ y_n + h\, f\left(x_n, y_n\right)\right)\right]$$

Let us consider $k_1 = h\, f\left(x_n, y_n\right)$ *and* $k_2 = h\, f\left(x_{n+1}, y_n + k_1\right)$ then the above equation takes the form

$$y_{n+1} = y_n + \frac{k_1 + k_2}{2}$$

This is the Modified Euler's formulae.

Local truncation Error due to the approximation:

Now the Modified Euler's formulae is

$$y_{n+1} = y_n + \frac{k_1 + k_2}{2}$$

with $k_1 = h\, f\left(x_n, y_n\right)$ *and* $k_2 = h\, f\left(x_n + h, y_n + k_1\right)$

Expanding k_2 in Taylor Series, we get,

$$k_2 = h\left[f\left(x_n, y_n\right) + h\frac{\partial}{\partial x} f\left(x_n, y_n\right) + k_1 \frac{\partial}{\partial y} f\left(x_n, y_n\right) + O(h^2, {k_1}^2)\right]$$

$$\textit{Since, } k_1 = h\, f\left(x_n, y_n\right) = O\left(h\right)$$

We can write

$$\therefore \quad \frac{1}{2}\left(k_1 + k_2\right) = h\, f\left(x_n, y_n\right) + \frac{h^2}{2}\left[\frac{\partial}{\partial x} f\left(x_n, y_n\right) + f\left(x_n, y_n\right)\frac{\partial}{\partial y} f\left(x_n, y_n\right)\right] + O(h^3) \quad .....(4)$$

And the Taylor series of $y_{n+1} = y\left(x_n + h\right)$ is

$$y\left(x_n + h\right) = y\left(x_n\right) + h\, y'\left(x_n\right) + \frac{h^2}{2} y''\left(x_n\right) + O\left(h^3\right)$$

Substitute

$$y'(x_n) = f(x_n, y_n) \quad \&$$

$$y'' = \frac{\partial}{\partial x} f(x_n, y_n) + \frac{d}{dx} y(x_n) \frac{\partial}{\partial y} f(x_n, y_n)$$

$$= \frac{\partial}{\partial x} f(x_n, y_n) + f(x_n, y_n) \frac{\partial}{\partial y} f(x_n, y_n)$$

We get

$$y(x_n + h) = y(x_n) + h\, f(x_n, y_n) + \frac{h^2}{2}\left[\frac{\partial}{\partial x} f(x_n, y_n) + f(x_n, y_n) \frac{\partial}{\partial y} f(x_n, y_n)\right] + O(h^3)$$

.....(5)

So from equation (4) and (5) it is clear that,

$$y(x_{n+1}) = y(x_n) + \frac{1}{2}(k_1 + k_2) + O(h^3)$$

So the local truncation error is $O(h^3)$

Hence the modified Euler method is second order accurate.

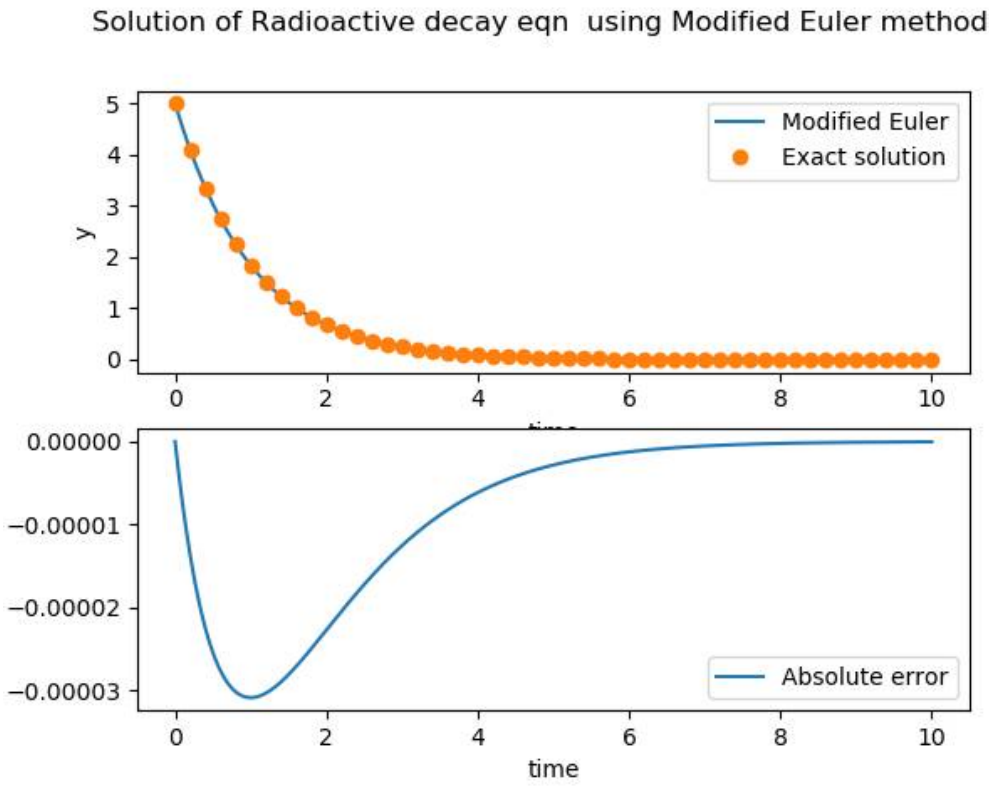
If we solve the differential equations that I shown in the example 1 to example-3 using Modified Euler method instead of Euler method then the error of the results will obviously be smaller. It can be checked from the error graph.

Python Code using Modified Euler method to solve ODEs.

- **Example 4: Radioactive decay (same as in example -1)**

```
# Solution of Radio-active decay equation (1st order ODE)
# Modified Euler method

import matplotlib.pyplot as plt
import numpy as np

# initial value of t, y
k=1
t=0.0
y=5.0
y0=y
tt=[]
yy=[]
tf=10 # final value of x
dt=0.01 # step length
```

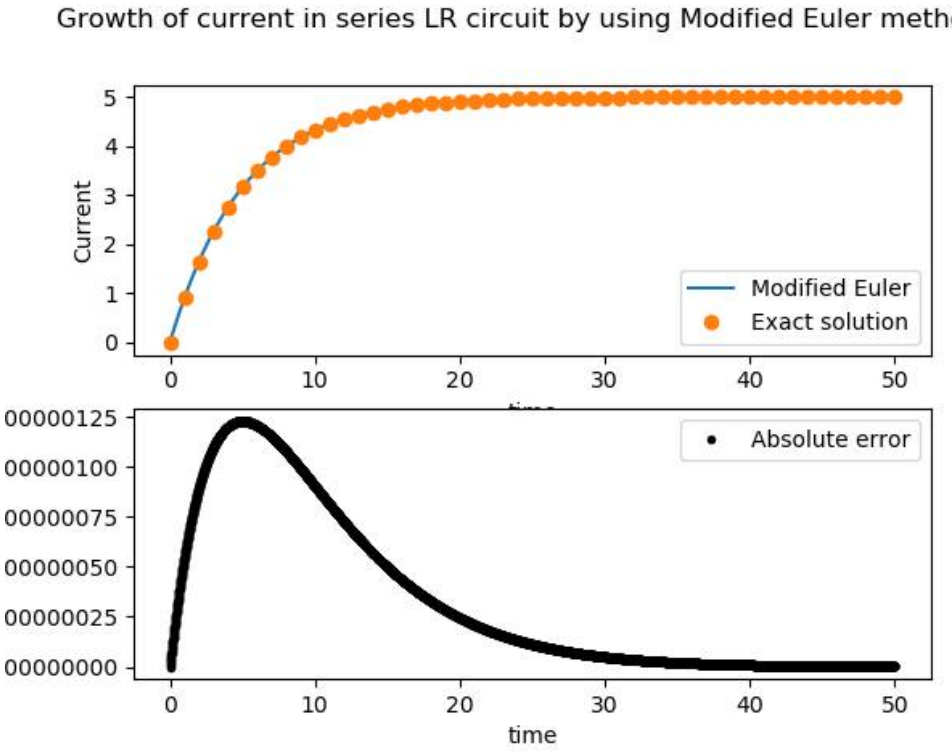
```
# defining function
def f(t,y,k):

     return -k*y


while t<=tf:

     tt.append(t)
     yy.append(y)
     k1=dt*f(t,y,k)
     k2=dt*f(t+dt,y+k1,k)
     y=y+0.5*(k1+k2)
     t+=dt

# Exact solution
tt=np.array(tt)
exact=y0*np.exp(-k*tt)
# Plotting of solution with exact result
plt.subplot(2,1,1)
plt.plot(tt,yy,'k--',label="dt=%.4f"%(dt))
plt.plot(tt, exact,'k',label="Exact solution")
plt.xlabel("time")
plt.ylabel("y")
plt.legend(loc='best')
plt.title("ODE by Modified Euler method")

# Calculation of Error and plotting
diff=exact-yy
plt.subplot(2,1,2)
plt.plot(tt,diff,'k.',label="Absolute error")
plt.xlabel("time")
plt.ylabel("Error")
plt.legend(loc='best')
plt.savefig("decay-m-euler-error.png")
```



➢ **Example 5: Growth of current in series LR circuit (same as in example -2)**

```
# Solution of LR circuit (1st order ODE)
# Modified Euler method
import matplotlib.pyplot as plt
import numpy as np

# initial value of t, i
e=1.0
r=0.2
l=1.0


t=0.0
i=0.0

tt=[]
ii=[]
tf=50.0 # final value of t
dt=0.01 # step length

# defining function
def f(t,i):

     return (e-i*r)/l
```

```
while t<=tf:
     tt.append(t)
     ii.append(i)
     k1=dt*f(t,i)
     k2=dt*f(t+dt,i+k1)
     i=i+0.5*(k1+k2)
     t+=dt


# Exact solution
tt=np.array((tt))
ii=np.array((ii))
exact=(e/r)*(1.0-np.exp(-r/l*tt))
# Plotting of solution with exact result
plt.subplot(2,1,1)
plt.plot(tt,ii,label="Modified Euler")
plt.plot(tt[::100], exact[::100],'o',lw='1',label="Exact
solution")
plt.xlabel("time")
plt.ylabel("Current")
plt.legend(loc='best')

# Calculation of Error and plotting
diff=exact-ii

plt.subplot(2,1,2)
plt.plot(tt,diff,'k.',label="Absolute error")
plt.xlabel("time")
plt.ylabel("Error")
plt.legend(loc='best')
plt.suptitle("Growth of current in series LR circuit by using
Modified Euler method")
plt.savefig("LR-m-euler-error.png")
```

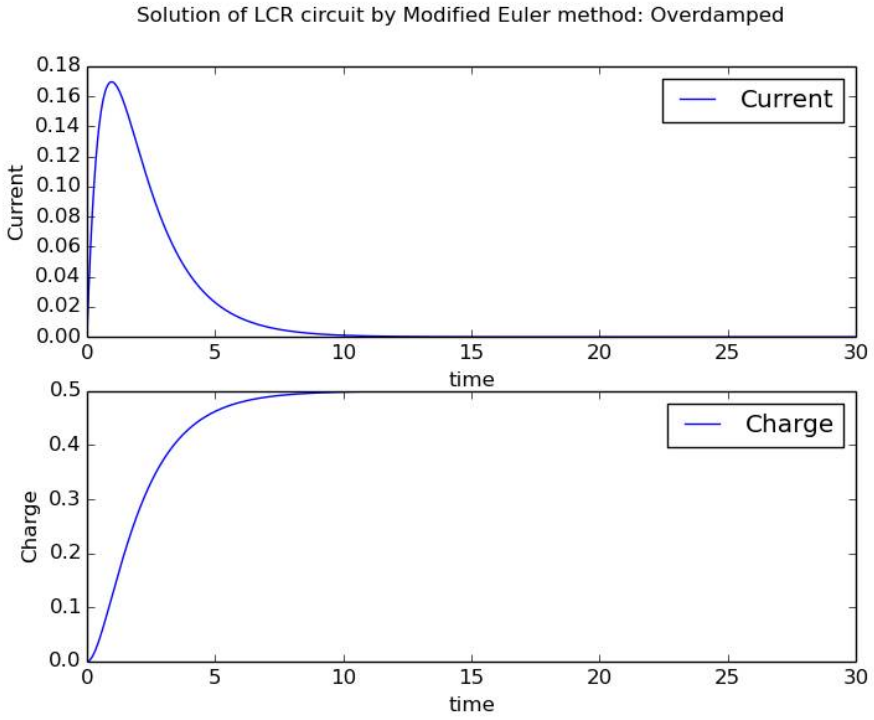


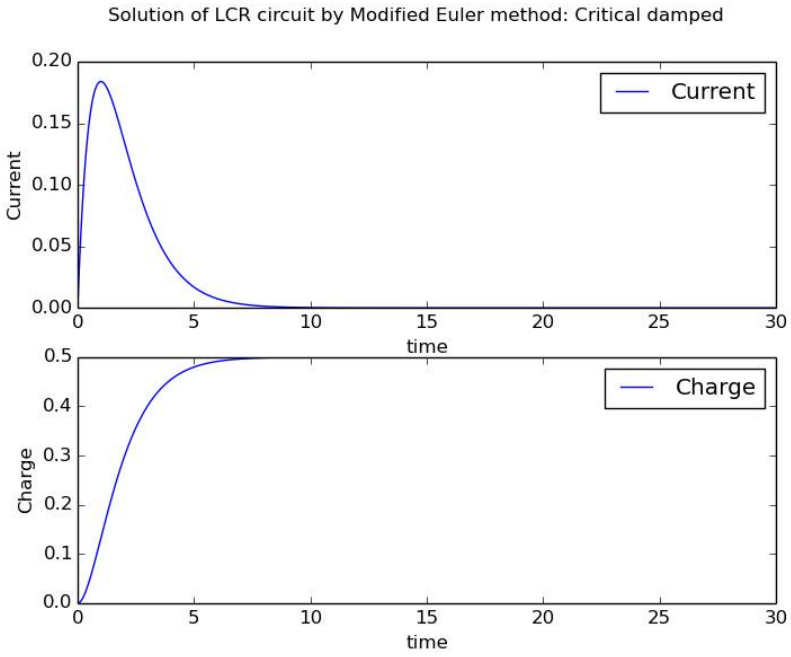
➢ **Example 6: Charging of capacitor in series RLC circuit (same as in example -3)**

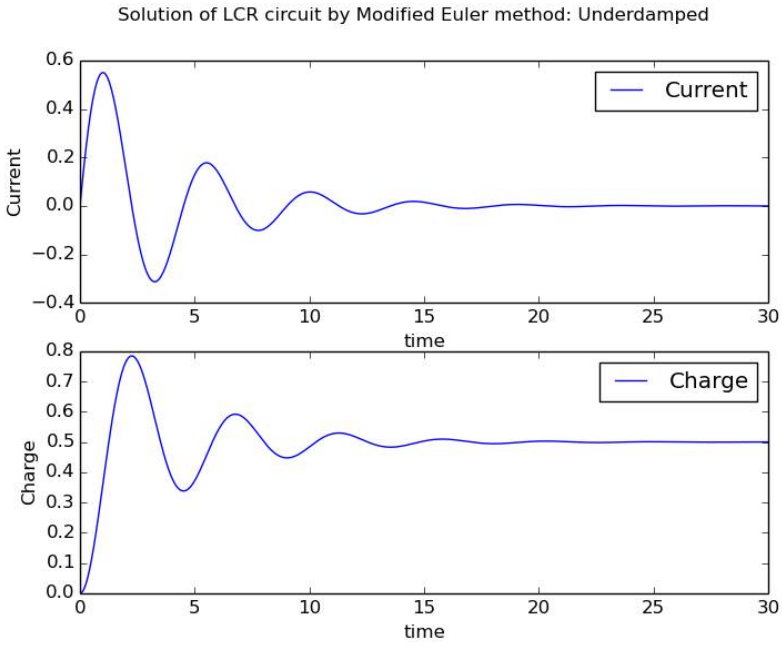
```
# Solution of LCR circuit (2nd order ODE)
# Modified Euler method

import matplotlib.pyplot as plt
import numpy as np
"""
# initial value of t, i
##########################################
###### Case-1, Overdamped case R>2*sqrt(L/C)
e=1.0
l=2.0
c=0.5
r=4.5
##########################################
"""
#"""
##########################################
###### Case-2, Critical damped case R=2*sqrt(L/C)
e=1.0
l=2.0
c=0.5
```

```
r=4
##############################################
#"""
"""
##########################################
###### Case-3, Underdamped case R<2*sqrt(L/C)
e=1.0
l=1.0
c=0.5
r=0.5
##########################################
"""
t=0.0
i=0.0
q=0.0

tf=30.0 # final value of t
dt=0.01 # step length


tt=[]
ii=[]
qq=[]

# defining function

def f1(q,i,t):
     return i #  dq/dt=i ..> f1(q,i,t)
def f2(q,i,t):

     return (e-i*r-q/c)/l  # di/dt=(e-ri-q/c)/l ..> f2(q,i,t)


while t<=tf:
     tt.append(t)
     qq.append(q)
     ii.append(i)

     p1=dt*f1(q,i,t) # dq at initial point
     p2=dt*f2(q,i,t) # di at initial point
     q1=dt*f1(q+p1,i+p2,t+dt) # dq at end point
     q2=dt*f2(q+p1,i+p2,t+dt) # di at end point

     q+=0.5*(p1+q1)
     i+=0.5*(p2+q2)
     t+=dt


plt.subplot(2,1,1)
plt.plot(tt,ii,label="Current")
plt.xlabel("time")
plt.ylabel("Current")
```

```
plt.legend(loc='best')


plt.subplot(2,1,2)
plt.plot(tt,qq,label="Charge")
plt.xlabel("time")
plt.ylabel("Charge")
plt.legend(loc='best')
plt.suptitle("Solution of LCR circuit by Modified Euler method:
Critical damped")
plt.savefig("Critical damped.png")
```



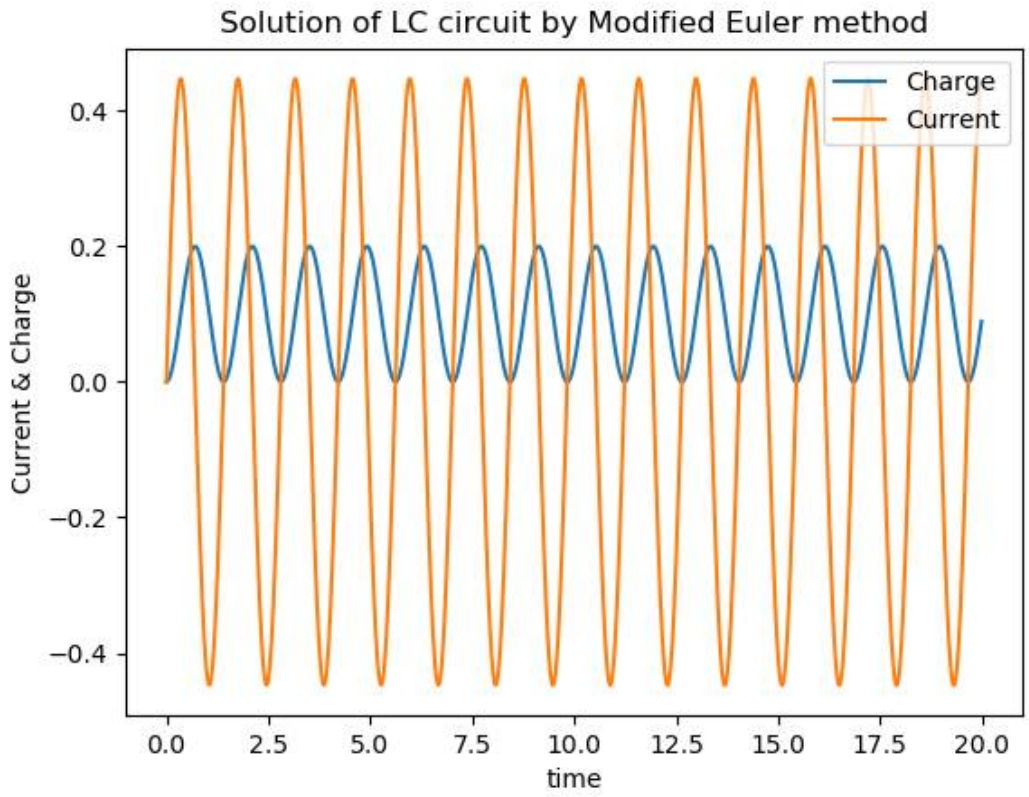




➢ Example 7: Solution of a series LC circuit

```
# Solution of LC circuit (1st order ODE)
# Modified Euler method
import matplotlib.pyplot as plt
import numpy as np
```

```
# initial value of t, i
e=1.0
l=0.5
c=0.1

t=0.0
i=0.0
q=0.0

tt=[]
ii=[]
qq=[]

tf=20.0 # final value of t
dt=0.01 # step length

# defining function

def f1(t,i,q):
     return i   # dq/dt=i

def f2(t,i,q):

     return (e-q/c)/l   # di/dt=(e-q/c)/l


while t<=tf:
     tt.append(t)
     ii.append(i)
     qq.append(q)

     p1=dt*f1(t,i,q)  # dq at initial point (t)
     p2=dt*f2(t,i,q)  # di at initial point (t)
     q1=dt*f1(t+dt,i+p2,q+p1) # dq at end point (t+dt)
     q2=dt*f2(t+dt,i+p2,q+p1) # di at end point (t+dt)
     q=q+0.5*(p1+q1)
     i=i+0.5*(p2+q2)
     t+=dt
plt.plot(tt,qq,label="Charge")
plt.plot(tt,ii,label="Current")
plt.xlabel("time")
plt.ylabel("Current & Charge")
plt.legend(loc='best')
plt.title("Solution of LC circuit by Modified Euler method")
plt.savefig("LC-m-euler.png")
```



3. Second Order Runge-Kutta Method (RK2 Method)

General 2nd order Runge-Kutta method takes the form

$$k_1 = h\, f\left(x_n, y_n\right)$$
$$k_2 = h\, f\left(x_n + \alpha h, y_n + \beta k_1\right)$$
$$y_{n+1} = y_n + a_1 k_1 + a_2 k_2 \qquad \ldots\ldots\ldots(6)$$

Similar to the analysis as done in Modified Euler method it can be shown that for this RK2 method

$$a_1 + a_2 = 1$$

&

$$\alpha a_2 = \beta a_2 = \frac{1}{2}$$

Since we have 3 equations and 4 unknowns a_1, a_2, α, β there are infinitely many solutions

The most popular are

❖ **Modified Euler method:**

$$a_1 = a_2 = \frac{1}{2},\ \ \alpha = \beta = 1$$

$$k_1 = h\, f\left(x_n, y_n\right),\ \ k_2 = h\, f\left(x_n + h, y_n + k_1\right)$$

and

$$y_{n+1} = y_n + \frac{1}{2}\left(k_1 + k_2\right)$$

❖ **Midpoint Method:**

$$a_1 = 0, a_2 = 1,\ \ \alpha = \beta = \frac{1}{2}$$

$$k_1 = h\, f\left(x_n, y_n\right),\ \ k_2 = h\, f\left(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}\right)$$

and

$$y_{n+1} = y_n + k_2$$

❖ **Heun's Method:**

$$a_1 = \frac{1}{4}, a_2 = \frac{3}{4},\ \ \alpha = \beta = \frac{2}{3}$$

$$k_1 = h\, f\left(x_n, y_n\right),\ \ k_2 = h\, f\left(x_n + \frac{2h}{3}, y_n + \frac{2k_1}{3}\right)$$

and

$$y_{n+1} = y_n + \frac{\left(k_1 + 3k_2\right)}{4}$$

4. Fourth Order Runge-Kutta Method (RK4 method)

Schemes of the form (6) can be extended to higher order methods. The most widely used Runge-Kutta scheme is the 4th order scheme RK4 based on Simpson's rule.

$$y_{n+1} = y_n + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right)$$

with

$$k_1 = h\ f\left(x_n, y_n\right)$$

$$k_2 = h\ f\left(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}\right)$$

$$k_3 = h\ f\left(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}\right)$$

$$k_4 = h\ f\left(x_n + h, y_n + k_3\right)$$

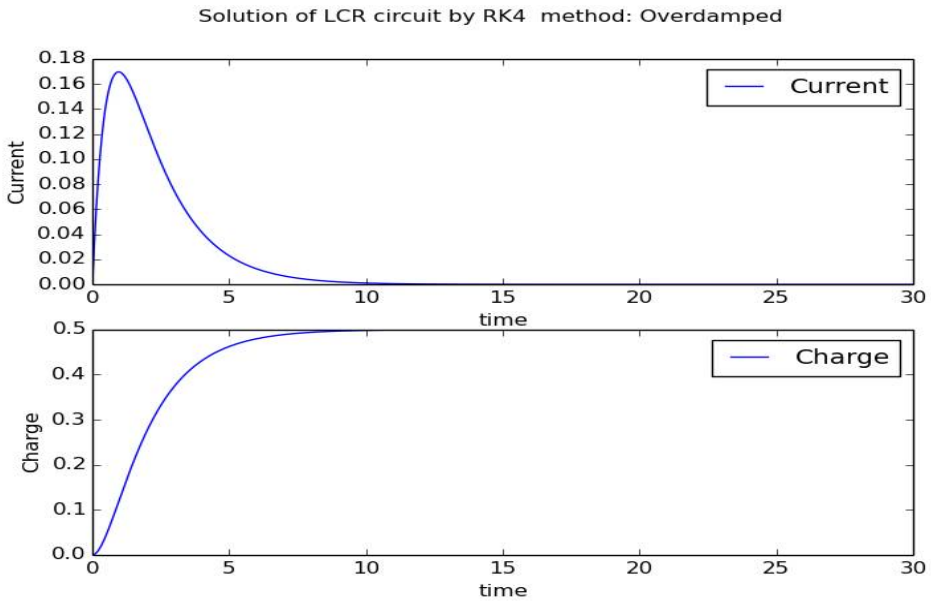
This scheme has local truncation error of order $O\left(h^5\right)$, which can be checked in the same way as the modified Euler scheme, but involves rather messy algebra.

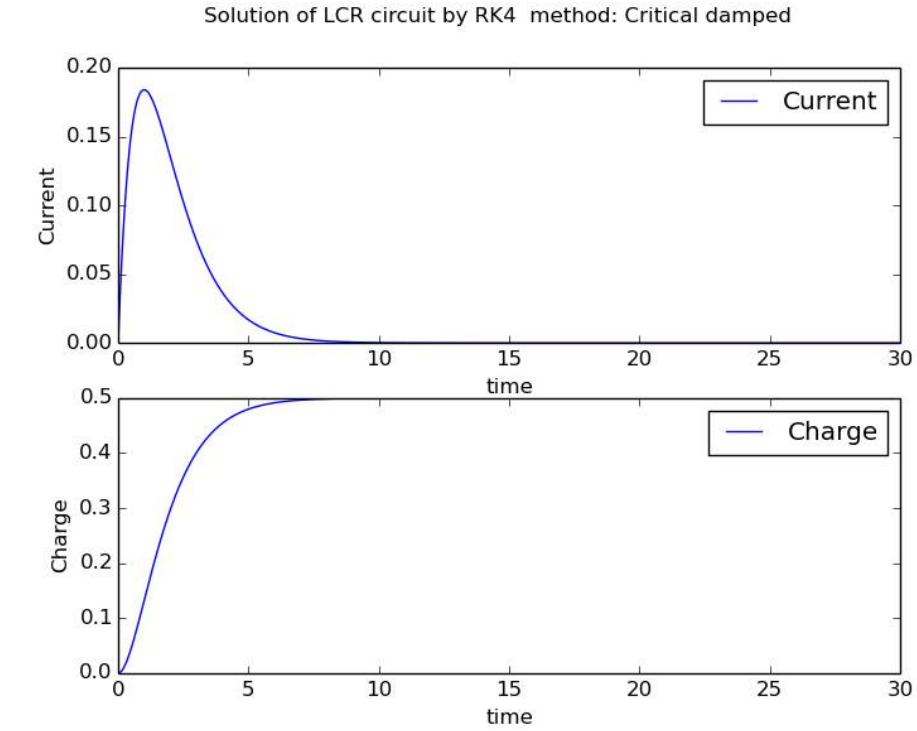
➢ **Example 8:**
Charging of capacitor in a series LCR circuit using RK4 method

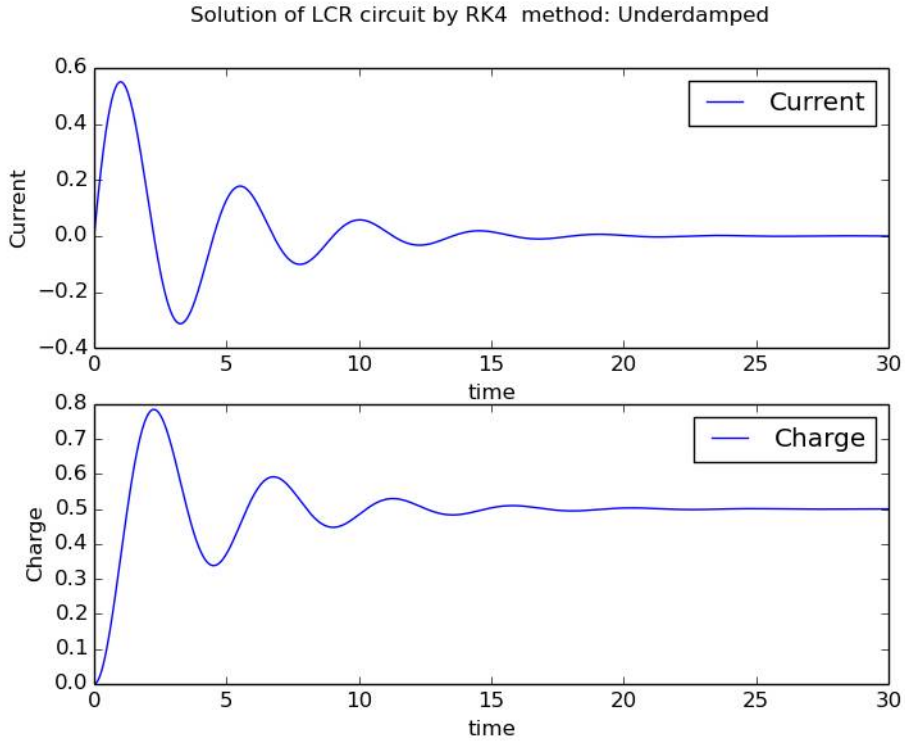
```
# Solution of LCR circuit (2nd order ODE)
# RK4 method

import matplotlib.pyplot as plt
import numpy as np
"""
# initial value of t, i
##########################################
###### Case-1, Overdamped damped case R>2*sqrt(L/C)
e=1.0
l=2.0
c=0.5
r=4.5
##########################################
"""
"""
##########################################
###### Case-2, Critical damped case R=2*sqrt(L/C)
```

```
e=1.0
l=2.0
c=0.5
r=4
##############################################
"""
#"""
##############################################
###### Case-3, Underdamped case R<2*sqrt(L/C)
e=1.0
l=1.0
c=0.5
r=0.5
##############################################
#"""
t=0.0
i=0.0
q=0.0

tf=30.0 # final value of t
dt=0.01 # step length


tt=[]
ii=[]
qq=[]


# defining function

def f1(q,i,t):
     return i #  dq/dt=i ..> f1(q,i,t)
def f2(q,i,t):

     return (e-i*r-q/c)/l  # di/dt=(e-ri-q/c)/l ..> f2(q,i,t)


while t<=tf:
     tt.append(t)
     qq.append(q)
     ii.append(i)

     p1=dt*f1(q,i,t) # dq at initial point
     p2=dt*f2(q,i,t) # di at initial point
     q1=dt*f1(q+0.5*p1,i+0.5*p2,t+0.5*dt)
     q2=dt*f2(q+0.5*p1,i+0.5*p2,t+0.5*dt)
     r1=dt*f1(q+0.5*q1,i+0.5*q2,t+0.5*dt)
     r2=dt*f2(q+0.5*q1,i+0.5*q2,t+0.5*dt)
     s1=dt*f1(q+r1,i+r2,t+dt)
     s2=dt*f2(q+r1,i+r2,t+dt)

     q+=(p1+2*q1+2*r1+s1)/6.0
```

```
        i+=(p2+2*q2+2*r2+s2)/6.0
        t+=dt


plt.subplot(2,1,1)
plt.plot(tt,ii,label="Current")
plt.xlabel("time")
plt.ylabel("Current")
plt.legend(loc='best')


plt.subplot(2,1,2)
plt.plot(tt,qq,label="Charge")
plt.xlabel("time")
plt.ylabel("Charge")
plt.legend(loc='best')
plt.suptitle("Solution of LCR circuit by RK4  method:
Underdamped")
plt.savefig("Underdamped.png")
```







- **RK4 Method Using Numpy:**

```
# Solution of LCR circuit (2nd order ODE)
# RK4 method
# Using numpy ( It looks like 1st order ODE)

import matplotlib.pyplot as plt
import numpy as np
```

```
"""
# initial value of t, i
##########################################
###### Case-1, Overdamped damped case R>2*sqrt(L/C)
e=1.0
l=2.0
c=0.5
r=4.5
##########################################
"""
"""
##########################################
###### Case-2, Critical damped case R=2*sqrt(L/C)
e=1.0
l=2.0
c=0.5
r=4
##########################################
"""
#"""
##########################################
###### Case-3, Underdamped case R<2*sqrt(L/C)
e=1.0
l=1.0
c=0.5
r=0.5
##########################################
#"""
t=0.0
i=0.0
q=0.0

tf=30.0 # final value of t
dt=0.01 # step length


tt=[]
ii=[]
qq=[]


# defining function

def f(Q,t):  # Q=(q,i)
     q,i=Q
     f1=i
     f2=(e-i*r-q/c)/l
     return np.array([f1,f2])

def update(Q,t):
     k1=dt*f(Q,t) # dQ at initial point
     k2=dt*f(Q+0.5*k1,t+0.5*dt)
```

```
        k3=dt*f(Q+0.5*k2,t+0.5*dt)
        k4=dt*f(Q+k3,t+dt)
        return Q+1/6.0*(k1+2*k2+2*k3+k4),t+dt

Q=np.array([q,i])
while t<=tf:
        tt.append(t)
        qq.append(Q[0])
        ii.append(Q[1])
        Q,t=update(Q,t)

plt.subplot(2,1,1)
plt.plot(tt,ii,label="Current")
plt.xlabel("time")
plt.ylabel("Current")
plt.legend(loc='best')


plt.subplot(2,1,2)
plt.plot(tt,qq,label="Charge")
plt.xlabel("time")
plt.ylabel("Charge")
plt.legend(loc='best')
plt.suptitle("Solution of LCR circuit by RK4  method:
Underdamped")
plt.savefig("Underdamped-numpy.png")
```

---

5. Solving first and 2nd order ODEs using odeint() from Scipy.integrate module:

The general syntax of **scipy.integrate.odeint() :**

scipy.integrate.odeint(*func, y0, t, args=(), tfirst=False*)

This function solves the initial value problem for first order ode-s:

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

It should be noted that the in func(). the dependent variable is first and independent variable. If we want to write func() as func(t,y) then you need to use the argument tfirst=True.

y_0 : Array, initial condition on y (can be a vector)

args: tuple, optional, Extra arguments to pass to function.

Returns: y, array, shape (len(t), len(y_0))

Array containing the value of y for each desired time in t, with the initial value *y0* in the first row

➢ Example 9: Radioactive decay

```
# Solution of LR circuit (1st order ODE)
# Using odeint() from Scipy.integrate

import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import odeint

# initial value of t, i
e=1.0
r=0.2
l=1.0

t=0.0
i0=0.0
tf=50.0 # final value of t

t=np.linspace(t,tf,500)

# defining function
def f(i,t):
     return (e-i*r)/l


sol=odeint(f,i0,t)[:,0]

# Exact solution
exact=(e/r)*(1.0-np.exp(-r/l*t))


# Plotting of solution with exact result
plt.subplot(2,1,1)
plt.plot(t,sol,label='odeint')
plt.plot(t, exact,label='exact')
plt.xlabel("time")
plt.ylabel("Current")
plt.legend(loc='best')
#plt.show()

# Calculation of Error and plotting
diff=exact-sol

plt.subplot(2,1,2)
plt.plot(t,diff,'k.',label="Absolute error")
plt.xlabel("time")
plt.ylabel("Error")
plt.legend(loc='best')
```

```
plt.suptitle("Solution of LR ckt by Odeint from Scipy module")
plt.savefig("LR-odeint.png")
```

➢ Example 10: Series LCR circuit

```
# Solution of LCR circuit (2nd order ODE)
# Using odeint from scipy.integrate module

import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import odeint
"""
# initial value of t, i
##############################################
###### Case-1, Overdamped damped case R>2*sqrt(L/C)
e=1.0
l=2.0
c=0.5
r=4.5
##############################################
"""
"""
##############################################
###### Case-2, Critical damped case R=2*sqrt(L/C)
e=1.0
l=2.0
c=0.5
r=4
##############################################
"""
#"""
##############################################
###### Case-3, Underdamped case R<2*sqrt(L/C)
e=1.0
l=1.0
c=0.5
r=0.5
##############################################
#"""
t=0.0
i=0.0
q=0.0

tf=30.0 # final value of t

t=np.linspace(t,tf,500)

# defining function
```

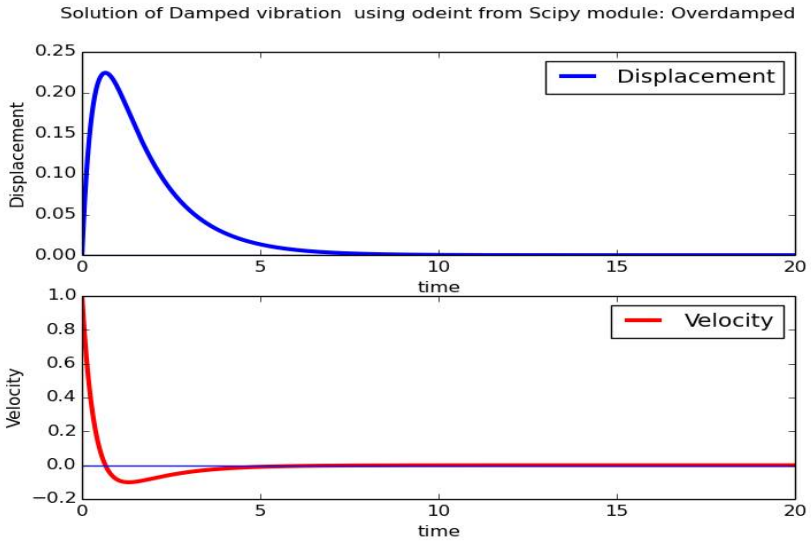
```
def f(Q,t):
     q,i=Q            # unpacking variables
     f1=i
     f2=(e-i*r-q/c)/l
     return np.array([f1,f2]) # packing functions

Q=np.array([q,i]) # packing initial values

sol=odeint(f,Q,t)
q,i=sol[:,0],sol[0:,1] # unpacking solutions q and i



plt.subplot(2,1,1)
plt.plot(t,i,label="Current")
plt.xlabel("time")
plt.ylabel("Current")
plt.legend(loc='best')


plt.subplot(2,1,2)
plt.plot(t,q,label="Charge")
plt.xlabel("time")
plt.ylabel("Charge")
plt.legend(loc='best')
plt.suptitle("Solution of LCR circuit using odeint from Scipy
module: underdamped")
plt.savefig("underdamped_odeint.png")
```

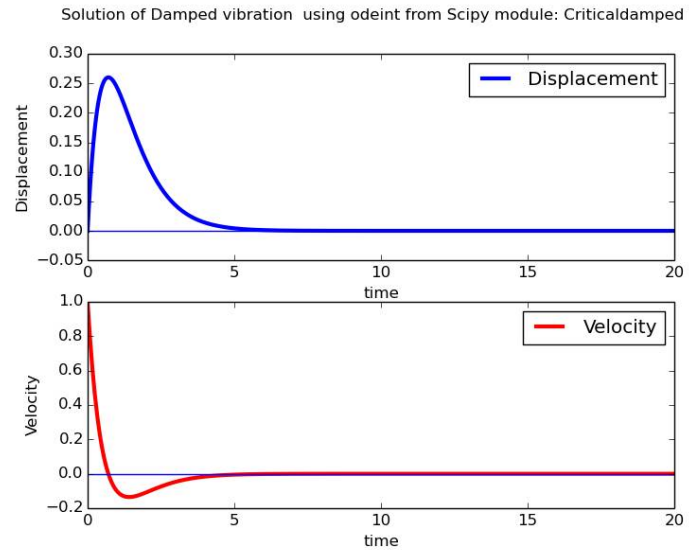
➢ Example 11: Damped Harmonic Oscillator

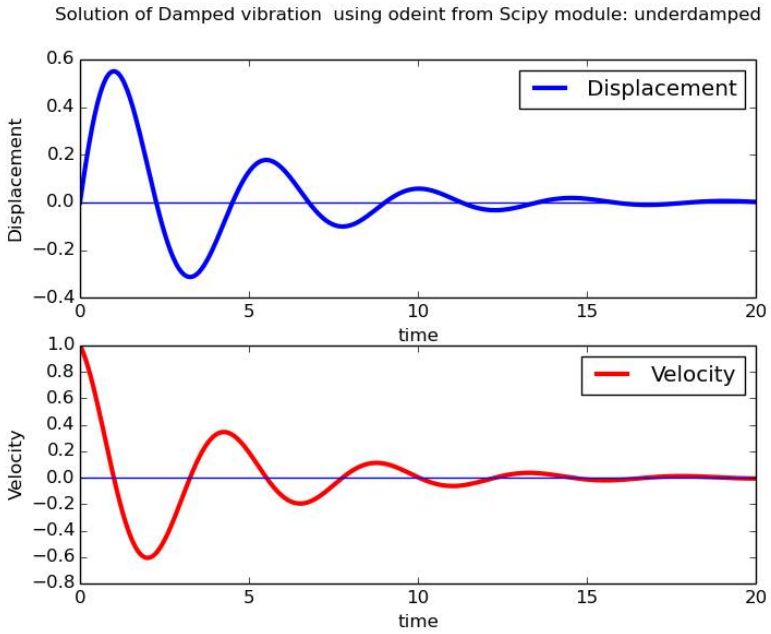
```
# Solution of damped vibration equationt (2nd order ODE)
# d^2x/dt^2+lambda*dx/dt+kx=0
# let dx/dt=v
# then dv/dt=-kx-lambda*x
# Using odeint from scipy.integrate module

import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import odeint
"""
#initial value of t, i
##########################################
###### Case-1, Overdamped case lambda>2*sqrt(k)
lam=3.5
k=2
```

```
##############################################
"""
#"""
##############################################
###### Case-2, Critical damped case lambda=2*sqrt(k)
lam=2*np.sqrt(2)
k=2
##############################################
#"""
"""
##############################################
###### Case-3, Underdamped case lambda<2*sqrt(k)
lam=0.5
k=2
##############################################
"""
t=0.0
v=1.0
x=0.0

tf=20.0 # final value of t

t=np.linspace(t,tf,500)

# defining function

def f(z,t):
     x,v=z            # unpacking variables
     f1=v
     f2=-k*x-lam*v
     return np.array([f1,f2]) # packing functions

z=np.array([x,v]) # packing initial values

sol=odeint(f,z,t)
x,v=sol[:,0],sol[0:,1] # unpacking solutions x and v



plt.subplot(2,1,1)
plt.plot(t,x,label="Displacement",lw=3)
plt.xlabel("time")
plt.ylabel("Displacement")
plt.legend(loc='best')
plt.axhline()


plt.subplot(2,1,2)
plt.plot(t,v,'r',label="Velocity",lw=3)
plt.xlabel("time")
plt.ylabel("Velocity")
plt.legend(loc='best')
```

```
plt.axhline()
plt.suptitle("Solution of Damped vibration  using odeint from
Scipy module: Criticaldamped")
plt.savefig("Critical-damped_odeint.png")
```



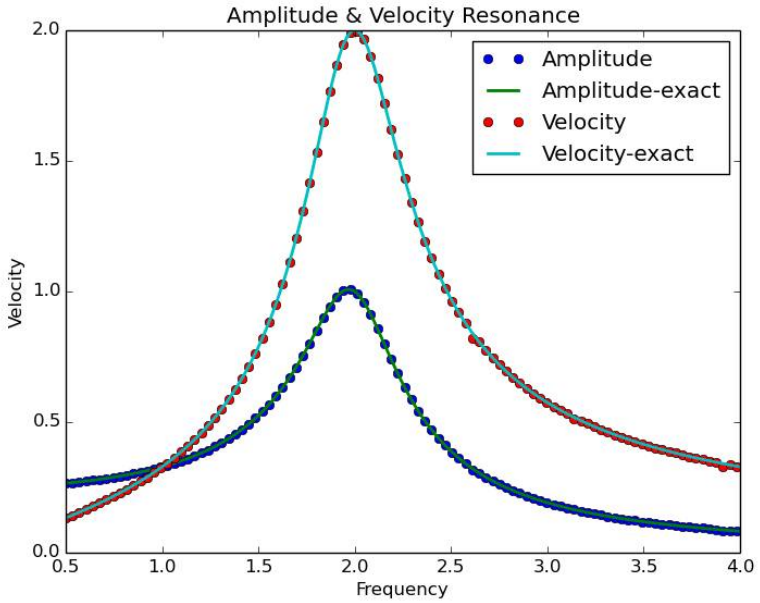




- **Example 12: Forced Vibration (Harmonic Oscillator)**
- **Amplitude resonance and Velocity resonance**

```
# Solution of forced vibration equationt (2nd order ODE)
# d^2x/dt^2+lambda*dx/dt+kx=Fcos(wt)
# let dx/dt=v
# then dv/dt=Fcos(wt)-kx-lambda*x
# Using odeint from scipy.integrate module
# To determine Amplitude and velocity resonace


import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import odeint


lam=0.5
k=4.0
F=1.0

tf=100.0 # final value of t

# defining function

def f(z,t):
     x,v=z            # unpacking variables
     f1=v
     f2=F*np.cos(w*t)-k*x-lam*v
     return np.array([f1,f2]) # packing functions
```

```
freq=np.linspace(0.5,4,200)
A,V=[],[]
for w in freq:
     t=0.0
     v=1.0
     x=0.0
     t=np.linspace(t,tf,500)
     z=np.array([x,v]) # packing initial values
     sol=odeint(f,z,t)
     x,v=sol[:,0],sol[0:,1] # unpacking solutions x and v
     amplitude=np.max(x[-200:])
     velocity=np.max(v[-200:])
     A.append(amplitude)
     V.append(velocity)

exact_a=F/np.sqrt((k-freq**2)**2+lam**2*freq**2)
exact_v=(F*freq)/np.sqrt((k-freq**2)**2+lam**2*freq**2)

plt.plot(freq[::2],A[::2],'o',label="Amplitude",lw=3)
plt.plot(freq,exact_a,label="Amplitude-exact",lw=2)
plt.plot(freq[::2],V[::2],'o',label="Velocity",lw=3)
plt.plot(freq,exact_v,label="Velocity-exact",lw=2)

plt.xlabel("Frequency")
plt.ylabel("Velocity")
plt.legend(loc='best')
plt.title("Amplitude & Velocity Resonance")
plt.savefig("Amplitude-velocity-resonace.png")
```



END