NUMERICAL METHODS

LAB RECORD

¶

MAT551B

Jeevan Koshy

1740256

37 Pages

5CMS

INDEX

Revision

11/06/2019

Importing all the libraries necessary for excecution

In [2]:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import optimize
import pylab as py
from scipy.misc import derivative
import math
from scipy.interpolate import interp1d
```

Lab 1: Basic Operations in Python for Scientific Computing

In [4]:

```
x=np.linspace(0,5,50) #50 values between 0 and 5
print(x)
```

```
[ 0.          0.10204082  0.20408163  0.30612245  0.40816327  0.51020408
  0.6122449   0.71428571  0.81632653  0.91836735  1.02040816  1.12244898
  1.2244898   1.32653061  1.42857143  1.53061224  1.63265306  1.73469388
  1.83673469  1.93877551  2.04081633  2.14285714  2.24489796  2.34693878
  2.44897959  2.55102041  2.65306122  2.75510204  2.85714286  2.95918367
  3.06122449  3.16326531  3.26530612  3.36734694  3.46938776  3.57142857
  3.67346939  3.7755102   3.87755102  3.97959184  4.08163265  4.18367347
  4.28571429  4.3877551   4.48979592  4.59183673  4.69387755  4.79591837
  4.89795918  5.        ]
```

What is the difference between piplot and pilab

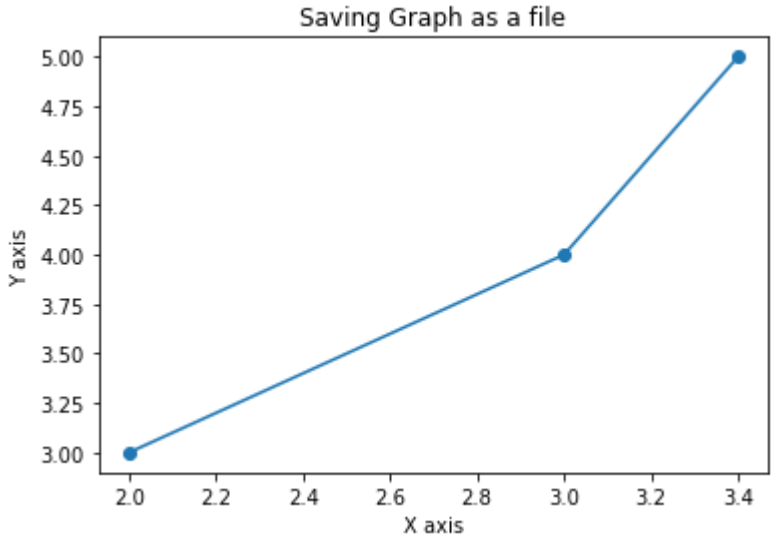
What is the difference between linspace and range, arange

In [5]:

```
x = [2,3,3.4]
y = [3,4,5]
py.plot(x,y,marker='o')
plt.savefig('plot.pdf')
plt.title("Saving Graph as a file")
plt.xlabel("X axis")
plt.ylabel("Y axis")
```

Out[5]:

```
<matplotlib.text.Text at 0x19167a7fe80>
```



In [6]:

```
x = np.arange(0,10,0.5) #decimal values
x
```

Out[6]:

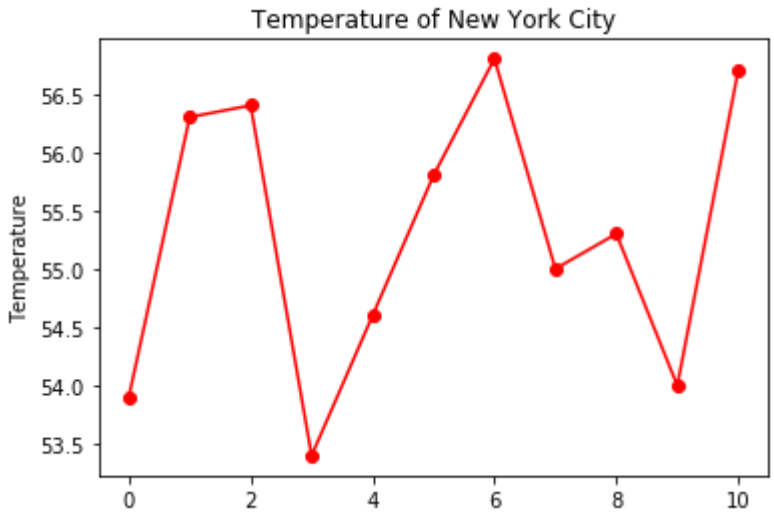
```
array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5,  5. ,
        5.5,  6. ,  6.5,  7. ,  7.5,  8. ,  8.5,  9. ,  9.5])
```

In [7]:

```
nyc_temp = [53.9,56.3,56.4,53.4,54.6,55.8,56.8,55.0,55.3,54.0,56.7]
py.plot(nyc_temp,marker='o',color='red')
plt.title("Temperature of New York City")
plt.ylabel("Temperature")
```

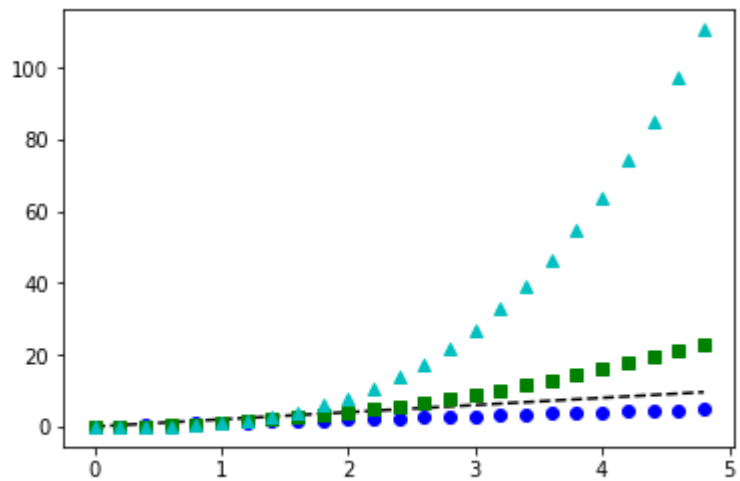
Out[7]:

```
<matplotlib.text.Text at 0x19167c585c0>
```



In [8]:

```
t = np.arange(0,5,0.2)

plt.plot(t,2*t,'k--',t,t,'bo',t,t**2,'gs',t,t**3,'c^')
plt.show()
```

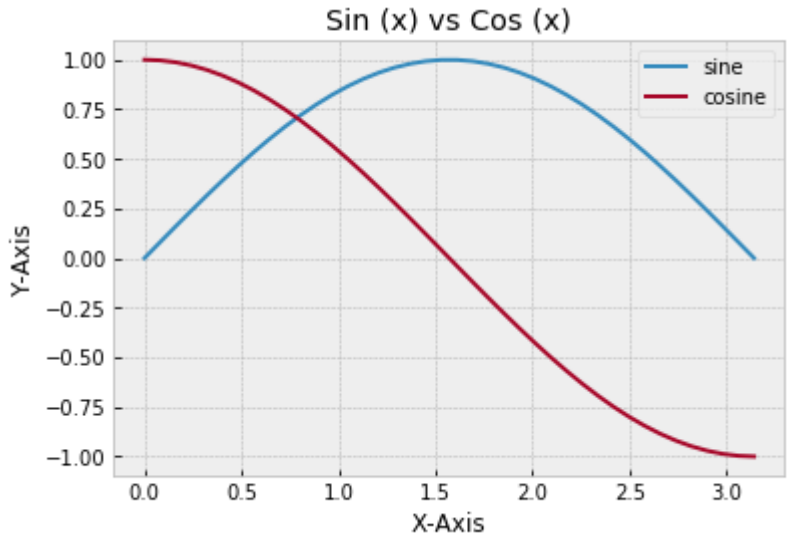


In [8]:

```
x=np.linspace(0,np.pi,40)
plt.style.use("bmh")
plt.plot(x,np.sin(x),x,np.cos(x))
plt.title("Sin (x) vs Cos (x)")
plt.xlabel("X-Axis")
plt.ylabel("Y-Axis")
plt.legend(["sine","cosine"])
```

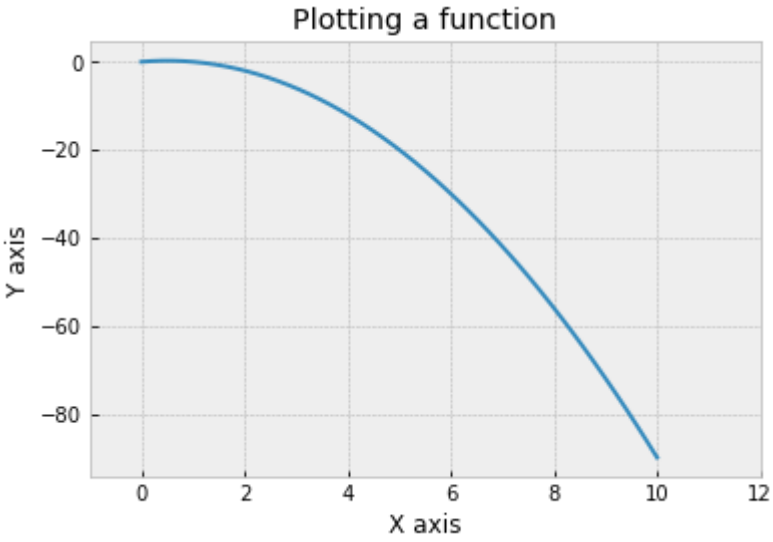
Out[8]:

```
<matplotlib.legend.Legend at 0x19167d4eeb8>
```



In [9]:

```
x = np.linspace(0,10,50)
y=x*(1-x)
plt.plot(x,y)
axis=plt.gca()
axis.set_xlim(-1,12)
plt.title("Plotting a function")
plt.xlabel("X axis")
plt.ylabel("Y axis")
plt.show()
```



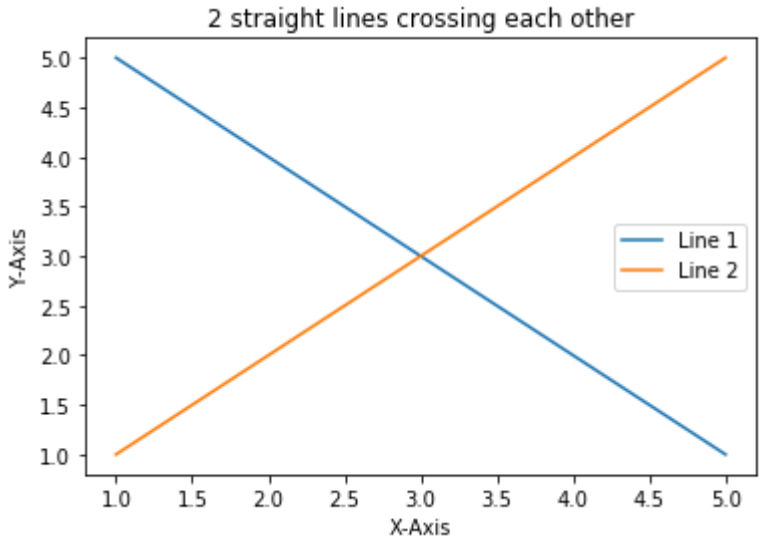
18th June 2019: Plots and Subplots

In [6]:

```
x1=np.array([1,2,3,4,5])
y1=np.array([5,4,3,2,1])
plt.plot(x1,y1)
x2=np.array([1,2,3,4,5])
y2=np.array([5,4,3,2,1])
y2=x2
plt.plot(x2,y2)
plt.legend(['Line 1', 'Line 2'])
plt.title("2 straight lines crossing each other")
plt.xlabel("X-Axis")
plt.ylabel("Y-Axis")
```

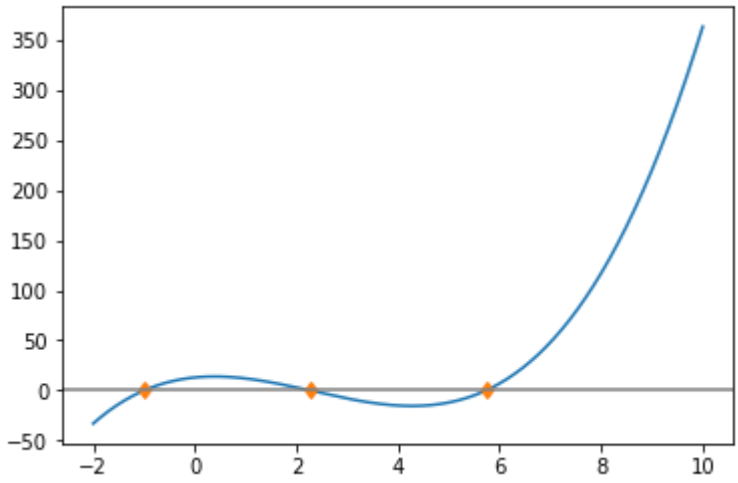
Out[6]:

```
<matplotlib.text.Text at 0x2236e48f390>
```



In [5]:

```
def fun(x,a=1,b=-7,c=5,d=13):
    return a*x**3 + b*x**2 + c*x + d

x = np.linspace(-2,10,1000)
sol = optimize.root(fun,[-2,2,10])

plt.plot(x,fun(x))
plt.plot(sol.x,fun(sol.x),'d')
plt.axhline(0,-2,2,color='gray')
plt.show()
```

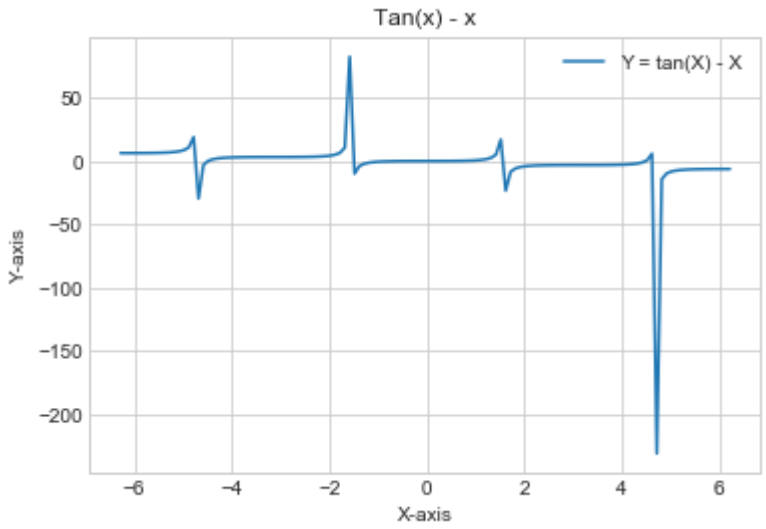


In [7]:

```
plt.style.use("seaborn-whitegrid")
x = np.arange(-2*np.pi,2*np.pi,0.1)
y = np.tan(x)-x
plt.plot(x,y)
plt.title('Tan(x) - x')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.legend(['Y = tan(X) - X'])
```

Out[7]:

```
<matplotlib.legend.Legend at 0x2236e5c4828>
```



Subplots of -

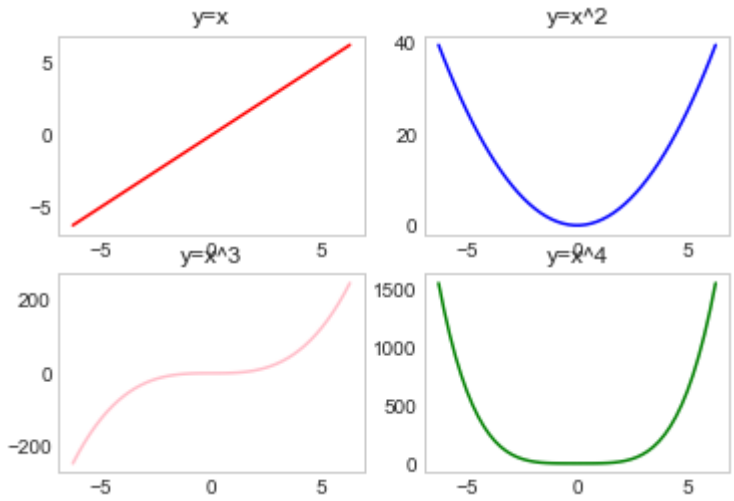
$y = x$
$y = x^2$
$y = x^3$
$y = x^4$

In [8]:

```
x=np.linspace(-2*np.pi,2*np.pi, 50)

plt.subplot(221)
plt.plot(x,x, color="red")
plt.title("y=x")
plt.grid()
print(end=" ")

plt.subplot(222)
plt.plot(x,x**2, color="blue")
plt.title("y=x^2")
plt.grid()


plt.subplot(223)
plt.plot(x,x**3, color="pink")
plt.grid()
plt.title("y=x^3")


plt.subplot(224)
plt.plot(x,x**4, color="green")
plt.grid()
plt.title("y=x^4")
```

Out[8]:

<matplotlib.text.Text at 0x2236e761710>



25th June 2019

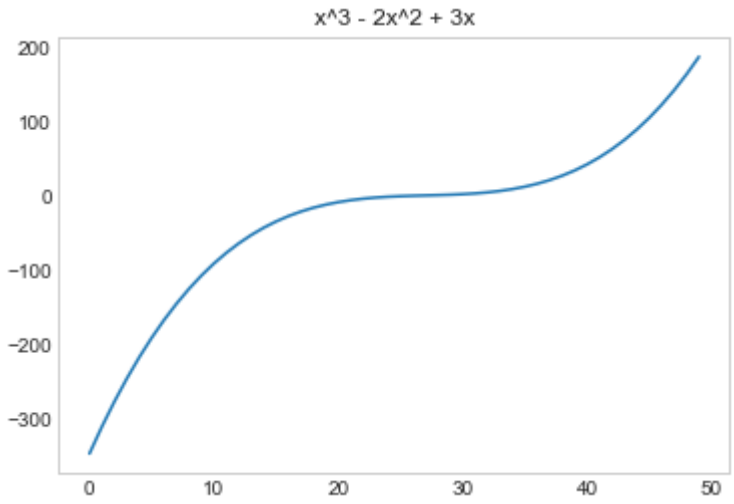
Solutions of Algebraic and Transcendal Equations

Find the root of the given equation

$$x^3 - 2x^2 + 3x - 1$$

In [9]:

```
y = x**3 - 2*x**2 + 3*x - 1
plt.plot(y)
plt.title('x^3 - 2x^2 + 3x')
plt.grid()
```

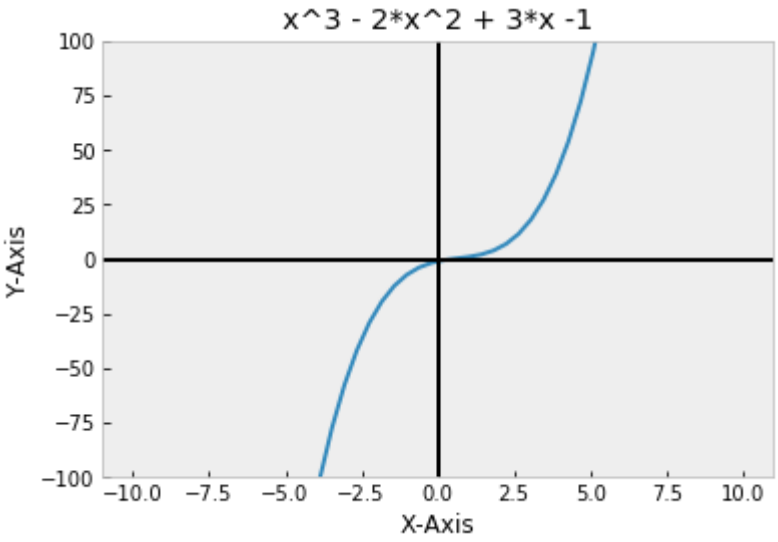


In [11]:

```
x= np.linspace(-10,10,50)
plt.plot(x,((x**3) - (2*x**2) +(3*x -1)))
plt.xlabel("X-Axis")
plt.ylabel("Y-Axis")
plt.ylim(-100,100)
plt.title("x^3 - 2*x^2 + 3*x -1")
plt.grid()
plt.axhline(y=0, color= "black")
plt.axvline(x=0, color= "black")
```

Out[11]:

```
<matplotlib.lines.Line2D at 0x18e26b84a58>
```



Bi - Section Method

Aim: To plot function graph and error graph of the given function using Bi - Section Method

In [12]:

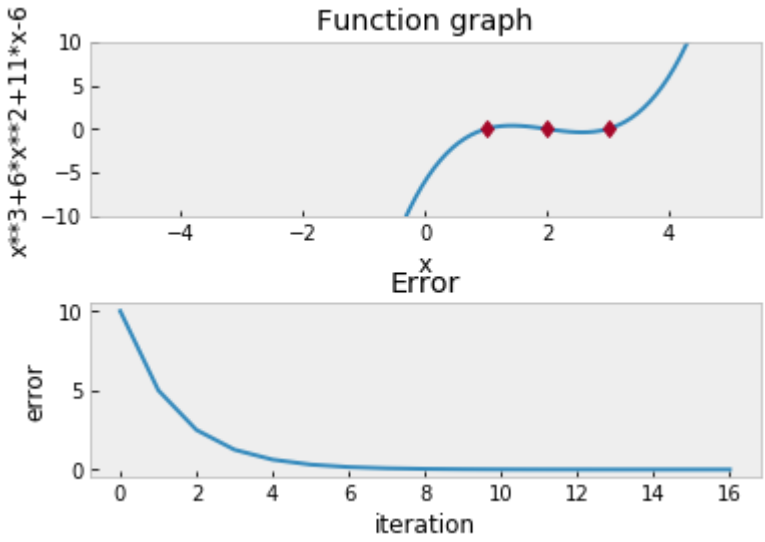
```
A=[]
B=[]
pres=float(input('enter precision limit '))
def fun(x):
    return x**3-6*x**2+11*x-6
while(True):
    a=float(input('enter lower limit: '))
    b=float(input('enter upper limit: '))
    if fun(a)*fun(b)<0:
        break
dash = '-' * 79

print(dash)
print('|{:^12s}|{:^12s}|{:^12s}|{:^12s}|{:^12s}|{:^12s}|'.format('iteration','a','b','c','f
print(dash)
i=0
while(True):
    if abs(b-a)<pres:
        break
    A.append(a)
    B.append(b)
    c=(a+b)/2
    i=i+1
    #print("Approx root",c)
    #print("f(c)",fun(c))
    print('|{:^12d}|{:^12.6f}|{:^12.6f}|{:^12.6f}|{:^12.6f}|{:^12.6f}|'.format(i,a,b,c,fun(
    if fun(c)==0:
        break
    else:
        if fun(a)*fun(c)<0:
            b=c
        else:
            a=c

print(dash)
print('By bisection method the root is {:.6f} at the {}th iteration'.format(c,i))

x=np.linspace(-5.0,5.0,100)

plt.subplot(2,1,1)
sol=optimize.root(fun,[0,2,3])
print(sol.x)
plt.plot(x,fun(x))
plt.plot(sol.x,fun(sol.x),'d')
plt.ylim(-10,10)
plt.gcf().subplots_adjust(hspace=0.5)
plt.title('Function graph')
plt.xlabel('x')
plt.ylabel('x**3+6*x**2+11*x-6')
plt.grid()
C=[]
for i,j in zip(A,B):
    C.append(abs((i-j)))
plt.subplot(2,1,2)
plt.plot(C)
plt.title('Error')
plt.xlabel('iteration')
plt.ylabel('error')
plt.grid()
```

```
plt.show()
plt.gcf().subplots_adjust(hspace=10)
```

```
enter precision limit 0.0001
enter lower limit: -5
enter upper limit: 5
```

iteration	a	b	c	f(c)	error
1	-5.000000	5.000000	0.000000	-6.000000	10.000000
2	0.000000	5.000000	2.500000	-0.375000	5.000000
3	2.500000	5.000000	3.750000	3.609375	2.500000
4	2.500000	3.750000	3.125000	0.298828	1.250000
5	2.500000	3.125000	2.812500	-0.276123	0.625000
6	2.812500	3.125000	2.968750	-0.059601	0.312500
7	2.968750	3.125000	3.046875	0.100445	0.156250
8	2.968750	3.046875	3.007812	0.015809	0.078125
9	2.968750	3.007812	2.988281	-0.023027	0.039062
10	2.988281	3.007812	2.998047	-0.003895	0.019531
11	2.998047	3.007812	3.002930	0.005885	0.009766
12	2.998047	3.002930	3.000488	0.000977	0.004883
13	2.998047	3.000488	2.999268	-0.001463	0.002441
14	2.999268	3.000488	2.999878	-0.000244	0.001221
15	2.999878	3.000488	3.000183	0.000366	0.000610
16	2.999878	3.000183	3.000031	0.000061	0.000305
17	2.999878	3.000031	2.999954	-0.000092	0.000153

```
By bisection method the root is 2.999954 at the 17th iteration
[ 1.  2.  3.]
```




```
<matplotlib.figure.Figure at 0x18e2674e5c0>
```

Bi - Section method was operated on the function and the approximate root is 2.999954. The error and function graph is also plotted for the function

Newton-Raphson Method

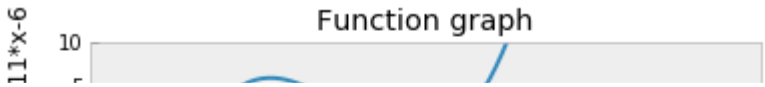
09/07/2019

In [13]:

```
X=list()
X1=list()


pres=float(input('enter precision limit: '))
def fun(x):
    return x**3+4*x**2+x-1
while(True):
    a=float(input('enter lower limit: '))
    b=float(input('enter upper limit: '))
    if fun(a)*fun(b)<0:
        break
x0=(a+b)/2
dash = '-' * 66
X.append(0)
X.append(x0)
X1.append(abs(X[-2]-X[-1]))
print(dash)
print('|{:^12s}|{:^12s}|{:^12s}|{:^12s}|{:^12s}|'.format('iteration','x','f(x)',"f'(x)",'er
print(dash)
i=0
while(True):
    print('|{:^12d}|{:^12.6f}|{:^12.6f}|{:^12.6f}|{:^12.6f}|'.format(i,X[-1],fun(X[-1]),der
    X.append(X[-1]-(fun(X[-1])/derivative(fun, X[-1])))

    X1.append(abs(X[-2]-X[-1]))
    i=i+1
    if fun(X[-1])==0:
        break
    if X1[-1]<pres:
        break
print(dash)
print('By Newton Raphson method the root is {:.6f} at the {}th iteration'.format(X[-1],i))
x=np.linspace(-5.0,5.0,100)

plt.subplot(2,1,1)
sol=optimize.root(fun,[0,2,3])
print(sol.x)
plt.plot(x,fun(x))
plt.plot(sol.x,fun(sol.x),'d')
plt.ylim(-10,10)
plt.gcf().subplots_adjust(hspace=0.5)
plt.title('Function graph')
plt.xlabel('x')
plt.ylabel('x**3+6*x**2+11*x-6')
plt.grid()


plt.subplot(2,1,2)
plt.plot(X1)
plt.title('Error vs Iteration')
plt.xlabel('iteration')
plt.ylabel('error')
plt.grid()
plt.show()
plt.gcf().subplots_adjust(hspace=10)
```

```
enter precision limit: 0.0001
```

```
enter lower limit: -5
enter upper limit: 5
-------------------------------------------------------------------
| iteration  |     x      |    f(x)    |   f'(x)    |   error    |
-------------------------------------------------------------------
|     0      |  0.000000  | -1.000000  |  2.000000  |  0.000000  |
|     1      |  0.500000  |  0.625000  |  6.750000  |  0.500000  |
|     2      |  0.407407  |  0.138952  |  5.757202  |  0.092593  |
|     3      |  0.383272  |  0.027163  |  5.506868  |  0.024135  |
|     4      |  0.378339  |  0.005058  |  5.456137  |  0.004933  |
|     5      |  0.377412  |  0.000931  |  5.446619  |  0.000927  |
|     6      |  0.377241  |  0.000171  |  5.444864  |  0.000171  |
-------------------------------------------------------------------
By Newton Raphson method the root is 0.377210 at the 7th iteration
[ 0.37720285  0.37720285  0.37720285]
```



Newton Rhapson method was operated on the function and the approximate root is 0.377210. The error and function graph is also plotted for the function

Gauss-Jordan Method

In [ ]:

```
def gauss_jordan(M, s):
    for i in range(0, s):
        count=0
        for j in range(i,s):
            if M[j,i]==0:
                count=count+1
        if count==(s-i):
            print("No unique solution for the system of linear equations")
            sys.exit()

        #Making ith non-zero initally
        non_zero_ele=i

        #print("Matrix is \n", M)
        #Look for a non-zero element if not at ith pos
        for j in range (i,s):
            if M[j,i]!=0:
                non_zero_ele=j
                break

        #Switch the ith row with non-zero element row if non-zero row isn't ith row itslef
        if non_zero_ele>i:
            temp=M[non_zero_ele].copy()
            M[non_zero_ele]=M[i,:]
            M[i,:]=temp

        #Make first element 1
        M[i]=M[i]/M[i,i]

        #Make lower elements 0
        if i!=(s-1):
            for j in range(i+1, s):
                M[j]-=M[i]*M[j,i]

        #Make elements above it 0 if row is not last and first row
        if i!=0:
            for j in range(0,i):
                M[j]-=M[i]*M[j,i]

    return M
```

Application Problems on Bi - Section Method & Newton - Rhaphson Method

16/07/2019

A bungee jumper with a mass of 68.1 kgs leaps from a stationary hot air balloon. Use the equation to compute velocity for the first 12s of free fall. Also determine the terminal velocity that will be

attained for an infinitely long cord. Use a drag coefficient of 0.25 kg/m.

$$\frac{d^2 f}{dx^2} - 5f = 0$$

$$\frac{dv}{dt} = g - \frac{Cd}{m} v^2$$

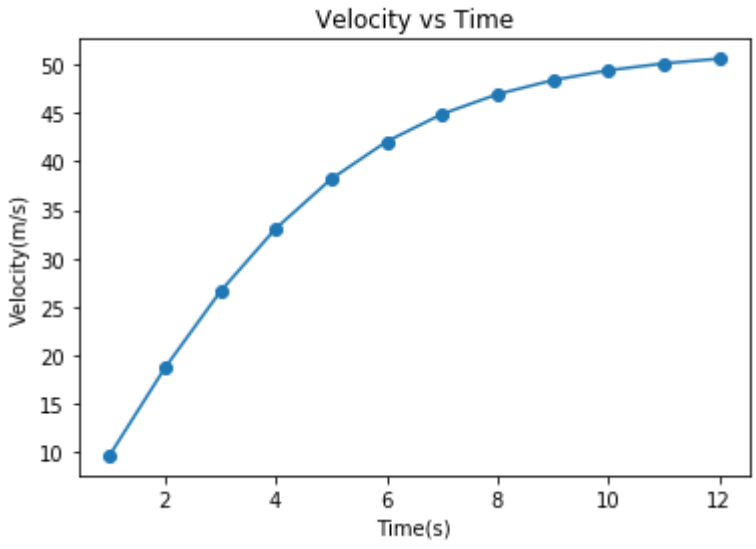
In [39]:

```
print("------------------------------")
print("|T (s)     |     Velocity(m/s)|")
print("------------------------------")
def velocity(m,t,cd):
    g = 9.8
    return (math.sqrt(g*m/cd)*np.tanh(math.sqrt(g*cd/m)*t))

velist=[]
for i in range(1,13):
    vel=velocity(68.1,i,0.25)
    print("|  {0} \t  | \t  {1}  ".format(i,round(vel,4)))
    velist.append(vel)

t = range(1,13)
plt.plot(t,velist,marker="o")
plt.title("Velocity vs Time")
plt.xlabel('Time(s)')
plt.ylabel('Velocity(m/s)')
```

```
------------------------------
|T (s)     |     Velocity(m/s)|
------------------------------
|  1       |       9.6841
|  2       |       18.711
|  3       |       26.5902
|  4       |       33.0832
|  5       |       38.1846
|  6       |       42.0446
|  7       |       44.883
|  8       |       46.9266
|  9       |       48.3755
|  10      |       49.3919
|  11      |       50.0993
|  12      |       50.5892
```

Out[39]:

```
<matplotlib.text.Text at 0x14531c091d0>
```



Use bisection method to determine the drag coefficient needed so that an $80 - kg$ bungee jumper has a velocity of $36m/s$ after $4s$ of

free fall. Note: The acceleration of gravity is $9.81 m/s^2$. Start with an initial guesses of $x(l) = 0.1$ and $x(u) = 0.2$ and iterate until the approximate relative error falls below 2.

$$v(t) = \sqrt{\frac{gm}{cd}} tanh \sqrt{\frac{gcd}{m}} t$$

$$f(cd) = \sqrt{\frac{9.81*80}{cd}} tanh(\sqrt{\frac{9.81cd}{80}} 4) - 36$$

In [16]:

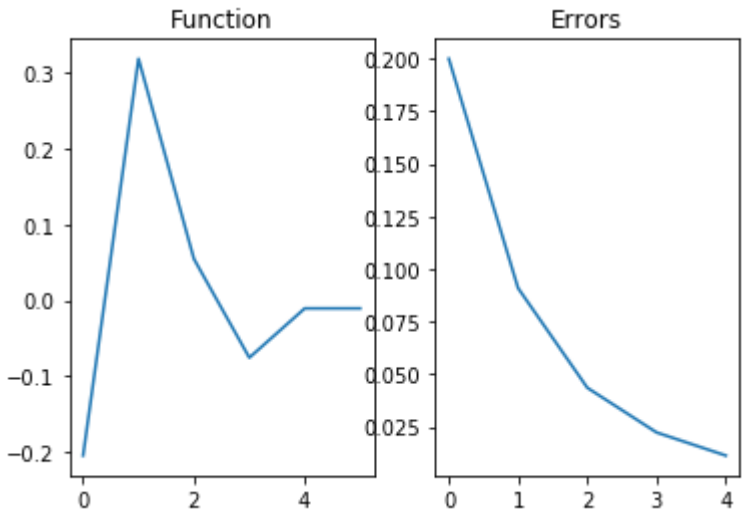
```
def fun(x)->float:
    return math.sqrt(9.81*80/x)*np.tanh(math.sqrt(9.81*x/80)*4)-36

def nextapprox(a, b)->float:
    return (a+b)/2

if __name__=="__main__":
    a=float(input("Enter lower limit: "))
    b=float(input("Enter upper limit: "))
    X=np.linspace(a-1,b+1,1000)

    print()
    print("a={0}".format(a))
    print("b={0}".format(b))
    neg=0.0
    pos=0.0
    count=0
    print("f(a)={0}\nf(b)={1}\n\n".format(round(fun(a),6),round(fun(b),6)))
    error=[]
    funlist=[]
    if fun(a)*fun(b)<0.0:
        dash = '-' * 113
        print(dash)
        print("x\t\t   a\t\t       b\t\t     Aprroximation\t\t f(approx)\tRel err")
        print(dash)
        print()
        if fun(a)<0.0:
            neg=a
            pos=b
        else:
            neg=b
            pos=a
        print("{0}\t\t{1:.6f}\t\t{2:.6f}\t\t{3:.6f}\t\t{4:.6f}\t{5:.6f}".format(count+1,rou
        while True:
            count=count+1
            if fun(neg)*fun(pos)<0.0:
                print()
                x0=nextapprox(neg, pos)
                funlist.append(fun(x0))
                if fun(x0)<0:
                    neg=round(x0, 6)
                else:
                    pos=round(x0, 6)
                x1=nextapprox(neg, pos)
                error.append(abs(pos-neg)/abs(pos+neg))
                #sol=optimize.root(fun,[1,4])
                print("{0}\t\t{1:.6f}\t\t{2:.6f}\t\t{3:.6f}\t\t{4:.6f}\t{5:.6f}".format(cou
            #if math.trunc(10**3 * x0) / 10**3==math.trunc(10**3 * x1) / 10**3:
            if(abs(pos-neg)/abs(pos+neg) < 0.02) :
                print()
                print("Approximate root is {0}".format(round(x1,6)))
                funlist.append(fun(x0))
                break
    else:
        print()
        print("Invalid interval entered")

plt.subplot(1,2,1)
plt.title("Function")
```

```
plt.plot(funlist)
plt.subplot(1,2,2)
plt.title("Errors")
plt.plot(error)
```

```
Enter lower limit: 0.1
Enter upper limit: 0.2

a=0.1
b=0.2
f(a)=0.860291
f(b)=-1.19738


--------------------------------------------------------------------------------
-------------------------------------
x                   a                   b                       Aprroximation
f(approx)       Rel err
--------------------------------------------------------------------------------
-------------------------------------

1               0.200000                0.100000                0.150000
-0.204516       -0.100000

2               0.150000                0.100000                0.125000
0.318407        0.050000

3               0.150000                0.125000                0.137500
0.054639        0.025000

4               0.150000                0.137500                0.143750
-0.075508       0.012500

5               0.143750                0.137500                0.140625
-0.010578       0.006250

6               0.140625                0.137500                0.139063
0.021995        0.003125

Approximate root is 0.139063
```

Out[16]:

```
[<matplotlib.lines.Line2D at 0x1f5d7370ba8>]
```



The volume of liquid V in a hollow horizontal cylinder of radius r and length L is related to the depth of the liquid h by

$$V = [r^2\cos^{-1}(\frac{r-h}{r}) - (r-h)\sqrt{2rh-h^2}]L$$

Determine h given r = 2m, L = $5m$; V = 8m^3 using Newton Raphson Method

$$8 = [4\cos^{-1}(\frac{2-h}{2}) - (2-h)\sqrt{4h-h^2}]5$$

$$f(h) = [4\cos^{-1}(\frac{2-h}{2}) - (2-h)\sqrt{4h-h^2}]5 - 8$$

In [6]:

```
X=list()
X1=list()
r=2
l=5
v=8

pres=float(input('enter precision limit: '))
def fun(h):
    return ((r**2)*math.acos((r-h)/r)-(r-h)*math.sqrt(2*r*h-h*h))*l-v
def derv(h):
    return (((r**2)*(1/np.sqrt(1-((2-h)/2)**2))+np.sqrt(2*r*h-h**2)-(r-h)*(2*r-2*h)/2*np.sq

x0=float(input("enter the initial guess: "))
dash = '-' * 66
X.append(0)
X.append(x0)
X1.append(abs(X[-2]-X[-1]))
print(dash)
print('|{:^12s}|{:^12s}|{:^12s}|{:^12s}|{:^12s}|'.format('iteration','h','f(h)',"f'(h)",'er
print(dash)
i=0
print('|{:^12d}|{:^12.6f}|{:^12.6f}|{:^12.6f}|{:^12.6f}|'.format(i,X[-1],fun(X[-1]),derv(X[
while(True):

    X.append(X[-1]-(fun(X[-1])/derv( X[-1])))

    X1.append(abs(X[-2]-X[-1]))
    i=i+1
    print('|{:^12d}|{:^12.6f}|{:^12.6f}|{:^12.6f}|{:^12.6f}|'.format(i,X[-1],fun(X[-1]),der
    if fun(X[-1])==0:
        break
    if X1[-1]<pres:
        break
print(dash)
print('By Newton Raphson method the root is {:.6f} at the {}th iteration'.format(X[-1],i))
```

```
enter precision limit: 0.001
enter the initial guess: 3
------------------------------------------------------------------
| iteration  |     h      |    f(h)    |   f'(h)    |   error    |
------------------------------------------------------------------
|     0      |  3.000000  | 42.548156  | 23.094011  |  3.000000  |
|     1      |  1.157611  |  7.080466  | 24.685103  |  1.842389  |
|     2      |  0.870779  |  2.096596  | 21.961037  |  0.286832  |
|     3      |  0.775310  |  0.553181  | 21.345652  |  0.095469  |
|     4      |  0.749395  |  0.146042  | 21.227002  |  0.025915  |
|     5      |  0.742515  |  0.038851  | 21.199693  |  0.006880  |
|     6      |  0.740682  |  0.010363  | 21.192730  |  0.001833  |
|     7      |  0.740193  |  0.002766  | 21.190895  |  0.000489  |
------------------------------------------------------------------
By Newton Raphson method the root is 0.740193 at the 7th iteration
```

You buy a 35,000 vehicle for nothing down at 8,500 per year for 7 years. Use the bisect function to determine the interest rate that you are paying. Employ initial guesses for the interest rate of 0.01

and 0.3 and a stopping criterion of 0.00005. The formula relating present worth P, annual payments A, number of years n, and interest rate i is

$$A = P\frac{i(1+i)^n}{(1+i)^n-1}$$

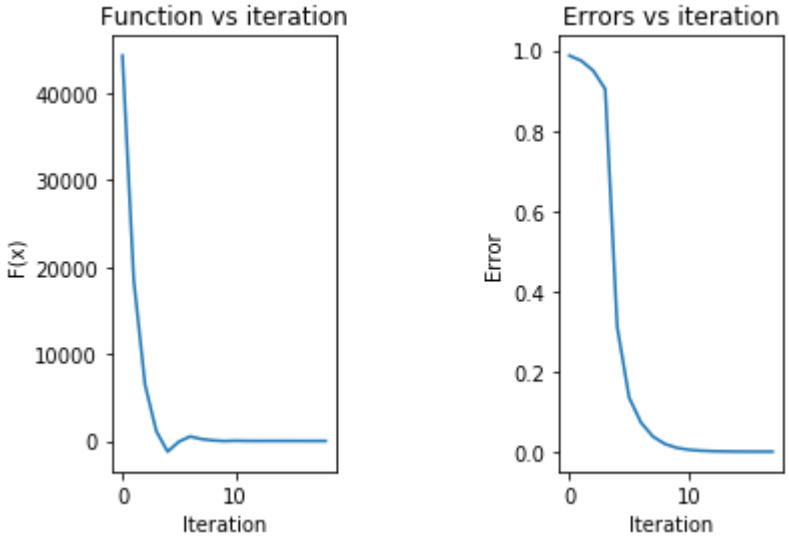
In [10]:

```
def fun(x)->float:
    return ((35000*(x*(1+x)**7))/(((1+x)**7)-1))-8500

def nextapprox(a, b)->float:
    return (a+b)/2

if __name__=="__main__":
    a=float(input("Enter lower limit: "))
    b=float(input("Enter upper limit: "))
    X=np.linspace(a-1,b+1,1000)

    print()
    print("a={0}".format(a))
    print("b={0}".format(b))
    neg=0.0
    pos=0.0
    count=0
    print("f(a)={0}\nf(b)={1}\n\n".format(round(fun(a),6),round(fun(b),6)))
    error=[]
    funlist=[]
    if fun(a)*fun(b)<0.0:
        dash = '-' * 113
        print(dash)
        print("x\t\t   a\t\t       b\t\t     Aprroximation\t\t f(approx)\tRel err")
        print(dash)
        print()
        if fun(a)<0.0:
            neg=a
            pos=b
        else:
            neg=b
            pos=a
        print("{0}\t\t{1:.6f}\t\t{2:.6f}\t\t{3:.6f}\t\t{4:.6f}\t{5:.6f}".format(count+1,rou
        while True:
            count=count+1
            if fun(neg)*fun(pos)<0.0:
                print()
                x0=nextapprox(neg, pos)
                funlist.append(fun(x0))
                if fun(x0)<0:
                    neg=round(x0, 6)
                else:
                    pos=round(x0, 6)
                x1=nextapprox(neg, pos)
                error.append(abs(pos-neg)/abs(pos+neg))
                #sol=optimize.root(fun,[1,4])
                print("{0}\t\t{1:.6f}\t\t{2:.6f}\t\t{3:.6f}\t\t{4:.6f}\t{5:.6f}".format(cou
            #if math.trunc(10**3 * x0) / 10**3==math.trunc(10**3 * x1) / 10**3:
            if(abs(pos-neg)/abs(pos+neg) < 0.00005) :
                print()
                print("Approximate root is {0}".format(round(x1,6)))
                funlist.append(fun(x0))
                break
    else:
        print()
        print("Invalid interval entered")

plt.subplot(1,2,1)
plt.title("Function vs iteration")
```

```
plt.plot(funlist)
plt.xlabel("Iteration")
plt.ylabel("F(x)")
plt.subplot(1,2,2)
plt.title("Errors vs iteration")
plt.xlabel("Iteration")
plt.ylabel("Error")
plt.plot(error)
plt.gcf().subplots_adjust(hspace=1)
plt.gcf().subplots_adjust(wspace=1)
```

```
Enter lower limit: 0.01
Enter upper limit: 3

a=0.01
b=3.0
f(a)=-3298.010098
f(b)=96506.409083


-------------------------------------------------------------------------
---------------------------------------
x                  a                   b                      Aprroximation
f(approx)       Rel err
-------------------------------------------------------------------------
---------------------------------------

1               0.010000               3.000000               1.505000
44260.241811    2.990000

2               0.010000               1.505000               0.757500
18534.476105    1.495000

3               0.010000               0.757500               0.383750
6472.570400     0.747500

4               0.010000               0.383750               0.196875
1126.769275     0.373750

5               0.010000               0.196875               0.103438
-1229.247973    0.186875

6               0.103438               0.196875               0.150156
-83.437363      0.093437

7               0.150156               0.196875               0.173515
514.070296      0.046719

8               0.150156               0.173515               0.161836
213.349214      0.023359

9               0.150156               0.161836               0.155996
64.460144       0.011680

10              0.150156               0.155996               0.153076
-9.619790       0.005840

11              0.153076               0.155996               0.154536
27.389073       0.002920
```

```
12              0.153076              0.154536              0.153806
8.876851        0.001460

13              0.153076              0.153806              0.153441
-0.373419       0.000730

14              0.153441              0.153806              0.153623
4.251229        0.000365

15              0.153441              0.153623              0.153532
1.932448        0.000182

16              0.153441              0.153532              0.153486
0.779484        0.000091

17              0.153441              0.153486              0.153464
0.196690        0.000045

18              0.153441              0.153464              0.153452
-0.082032       0.000023

19              0.153452              0.153464              0.153458
0.057329        0.000012

Approximate root is 0.153458
```



Type *Markdown* and LaTeX: α^2

Regular Falsi Method

02/09/2019

To find the approximate root of the given equation using Method of False Position/ Regula-Falsi Method

In [15]:

```
def fun(x)->float:
    return round((x*math.exp(x)-3), 6)

def nextapprox(a, b)->float:
    return (a*fun(b)-b*fun(a))/(fun(b)-fun(a))

if __name__=="__main__":
    a=float(input("Enter lower limit: "))
    b=float(input("Enter upper limit: "))
    print()
    print("x1={0}".format(a))
    print("x2={0}".format(b))
    neg=0.0
    pos=0.0
    count=0
    if fun(a)*fun(b)<0.0:
        if fun(a)<0.0:
            neg=a
            pos=b
        else:
            neg=b
            pos=a
        while True:
            count=count+1
            if fun(neg)*fun(pos)<0.0:
                print()
                x0=nextapprox(neg, pos)
                print("Approximation {1} is x{2}={0}".format(round(x0,6), count, count+2))
                print("f({0}) is {1}".format(round(x0, 6), round(fun(x0), 6)))
                if fun(x0)<0:
                    neg=round(x0, 6)
                else:
                    pos=round(x0, 6)
                print("Now root lies in ({0}, {1})".format(neg, pos))
                x1=nextapprox(neg, pos)
                print("Next approximation will be x={0}".format(round(x1, 6)))
            if math.trunc(10**3 * x0) / 10**3==math.trunc(10**3 * x1) / 10**3:
                print()
                print("Approximate root is {0}".format(math.trunc(10**3 * x1) / 10**3))
                break
    else:
        print()
        print("Invalid interval entered")
```

```
Enter lower limit: 1
Enter upper limit: 1.5

x1=1.0
x2=1.5

Approximation 1 is x3=1.035177
f(1.035177) is -0.085349
Now root lies in (1.035177, 1.5)
Next approximation will be x=1.045596

Approximation 2 is x4=1.045596
f(1.045596) is -0.025183
Now root lies in (1.045596, 1.5)
```

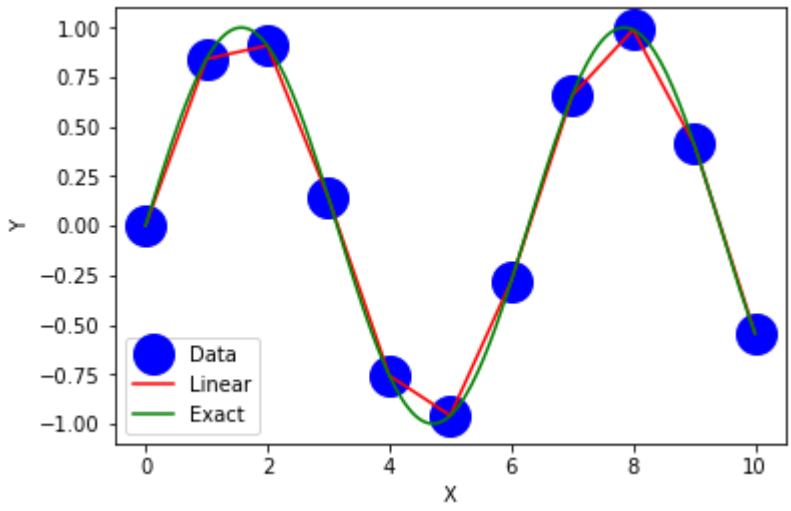
```
Next approximation will be x=1.048649

Approximation 3 is x5=1.048649
f(1.048649) is -0.007372
Now root lies in (1.048649, 1.5)
Next approximation will be x=1.049541

Approximation 4 is x6=1.049541
f(1.049541) is -0.002153
Now root lies in (1.049541, 1.5)
Next approximation will be x=1.049802

Approximate root is 1.049
```

Interpolation

10/09/2019

In [2]:

```
# make our tabular values
x_table = np.arange(11)
y_table = np.sin(x_table)

# linearly interpolate
x = np.linspace(0.,10.,201)

# here we create linear interpolation function
linear = interp1d(x_table,y_table,'linear')

# apply and create new array
y_linear = linear(x)

# plot results to illustrate
plt.ion()
plt.plot(x_table,y_table,'bo',markersize=20)
plt.plot(x,y_linear,'r')
plt.plot(x,np.sin(x),'g')
plt.legend(['Data','Linear','Exact'],loc='best')
plt.xlabel('X')
plt.ylabel('Y')
plt.show()
```



In [5]:

```
def lagrange(x,i,xm):
    """
    Evaluates the i-th Lagrange polynomial
    at xbased on grid data xm
    """
    n=len(xm)-1
    y=1
    for j in range(n+1):
        if i!=j:
            y*=(x-xm[j])/(xm[i]-xm[j])
    return y

def interpolation(x,xm ,ym):
    n=len(xm)-1
    lagrpoly=np.array([lagrange(x,i,xm) for i in range(n+1)])
    y = np.dot(ym ,lagrpoly)
    return y

# Example
xm = np.array([1,2,3,4,5,6])
ym = np.array([-3,0,-1,2,1,4])
xplot = np.linspace(0.9,6.1,100)
yplot = interpolation(xplot ,xm,ym)
```

In [6]:

```
xplot
```

Out[6]:

```
array([ 0.9       ,  0.95252525,  1.00505051,  1.05757576,  1.11010101,
        1.16262626,  1.21515152,  1.26767677,  1.32020202,  1.37272727,
        1.42525253,  1.47777778,  1.53030303,  1.58282828,  1.63535354,
        1.68787879,  1.74040404,  1.79292929,  1.84545455,  1.8979798 ,
        1.95050505,  2.0030303 ,  2.05555556,  2.10808081,  2.16060606,
        2.21313131,  2.26565657,  2.31818182,  2.37070707,  2.42323232,
        2.47575758,  2.52828283,  2.58080808,  2.63333333,  2.68585859,
        2.73838384,  2.79090909,  2.84343434,  2.8959596 ,  2.94848485,
        3.0010101 ,  3.05353535,  3.10606061,  3.15858586,  3.21111111,
        3.26363636,  3.31616162,  3.36868687,  3.42121212,  3.47373737,
        3.52626263,  3.57878788,  3.63131313,  3.68383838,  3.73636364,
        3.78888889,  3.84141414,  3.89393939,  3.94646465,  3.9989899 ,
        4.05151515,  4.1040404 ,  4.15656566,  4.20909091,  4.26161616,
        4.31414141,  4.36666667,  4.41919192,  4.47171717,  4.52424242,
        4.57676768,  4.62929293,  4.68181818,  4.73434343,  4.78686869,
        4.83939394,  4.89191919,  4.94444444,  4.9969697 ,  5.04949495,
        5.1020202 ,  5.15454545,  5.20707071,  5.25959596,  5.31212121,
        5.36464646,  5.41717172,  5.46969697,  5.52222222,  5.57474747,
        5.62727273,  5.67979798,  5.73232323,  5.78484848,  5.83737374,
        5.88989899,  5.94242424,  5.99494949,  6.04747475,  6.1       ])
```

In [7]:

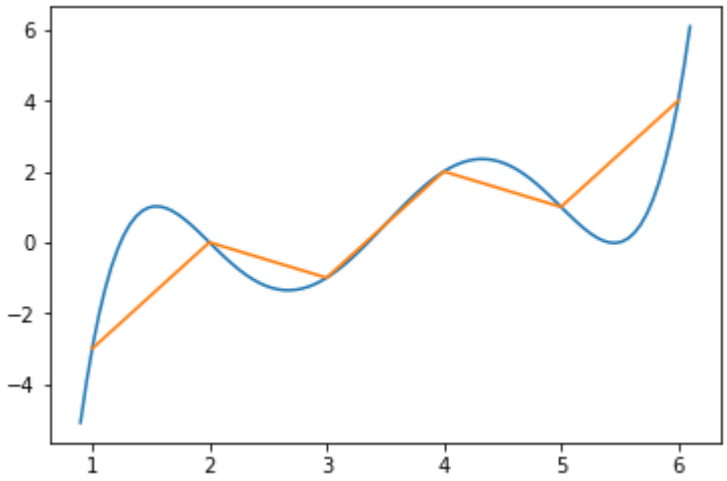
```
yplot
```

Out[7]:

```
array([-5.088336  , -3.91939949, -2.90943252, -2.04543428, -1.31501164,
       -0.70636638, -0.20828239,  0.18988713,  0.49823246,  0.72630016,
        0.88310586,  0.97714704,  1.01641584,  1.00841186,  0.96015492,
        0.8781979 ,  0.76863949,  0.63713702,  0.48891922,  0.32879903,
        0.16118642, -0.00989888, -0.18081452, -0.34828279, -0.50937777,
       -0.66151261, -0.80242667, -0.93017274, -1.04310429, -1.13986262,
       -1.21936411, -1.28078738, -1.32356056, -1.34734843, -1.35203968,
       -1.33773406, -1.30472967, -1.25351007, -1.18473155, -1.09921033,
       -0.99790974, -0.88192745, -0.75248267, -0.61090334, -0.45861338,
       -0.29711985, -0.12800016,  0.04711068,  0.2265329 ,  0.40855474,
        0.59144526,  0.7734671 ,  0.95288932,  1.12800016,  1.29711985,
        1.45861338,  1.61090334,  1.75248267,  1.88192745,  1.99790974,
        2.09921033,  2.18473155,  2.25351007,  2.30472967,  2.33773406,
        2.35203968,  2.34734843,  2.32356056,  2.28078738,  2.21936411,
        2.13986262,  2.04310429,  1.93017274,  1.80242667,  1.66151261,
        1.50937777,  1.34828279,  1.18081452,  1.00989888,  0.83881358,
        0.67120097,  0.51108078,  0.36286298,  0.23136051,  0.1218021 ,
        0.03984508, -0.00841186, -0.01641584,  0.02285296,  0.11689414,
        0.27369984,  0.50176754,  0.81011287,  1.20828239,  1.70636638,
        2.31501164,  3.04543428,  3.90943252,  4.91939949,  6.088336  ])
```

In [13]:

```
plt.plot(xplot,yplot)
plt.plot(xm,ym)
plt.show()
```



In [12]:

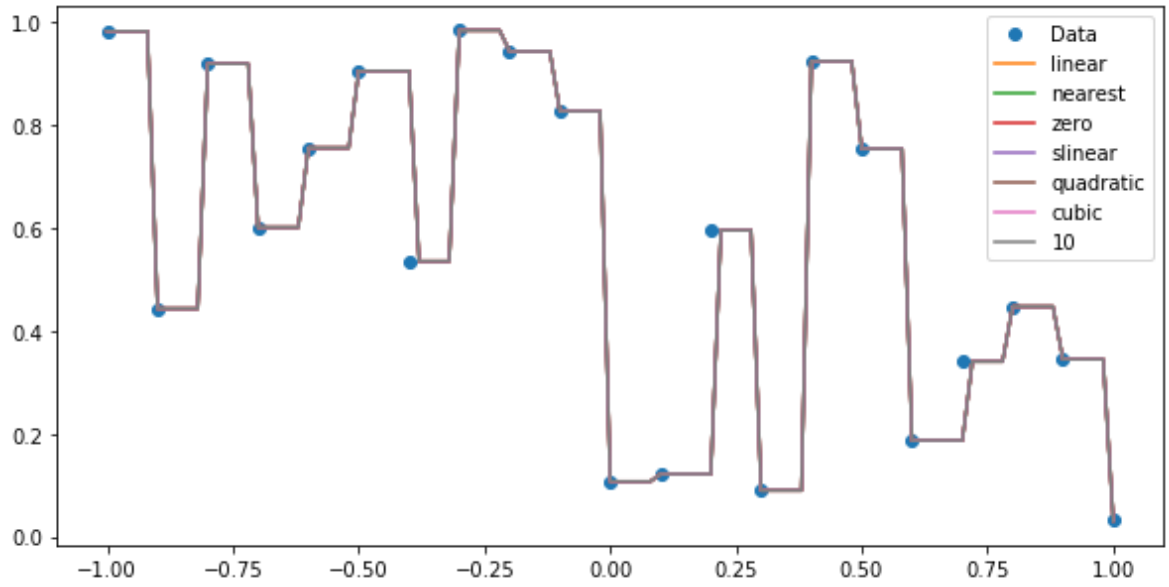
```
# Program to interpolate using
# newton forward interpolation
# calculating u mentioned in the formula
def u_cal(u, n):
    temp = u;
    for i in range(1, n):
        temp = temp * (u - i);
    return temp;

# calculating factorial of given number n
def fact(n):
    f = 1;
    for i in range(2, n + 1):
        f *= i;
    return f;

# Driver Code
# Number of values given
n = 4;
x = [ 45, 50, 55, 60 ];
# y[][] is used for difference table
# with y[][0] used for input
y = [[0 for i in range(n)]
        for j in range(n)];
y[0][0] = 0.7071;
y[1][0] = 0.7660;
y[2][0] = 0.8192;
y[3][0] = 0.8660;
# Calculating the forward difference table

for i in range(1, n):
    for j in range(n - i):
        y[j][i] = y[j + 1][i - 1] - y[j][i - 1];
    # Displaying the forward difference table

for i in range(n):
    print(x[i], end = "\t");
    for j in range(n - i):
        print(y[i][j], end = "\t");
    print("");

# Value to interpolate at
value = 52;
# initializing u and sum
sum = y[0][0];
u = (value - x[0]) / (x[1] - x[0]);

for i in range(1,n):
    sum = sum + (u_cal(u, i) * y[0][i]) / fact(i);

print("\nValue at", value,"is", round(sum, 6));
# This code is contributed by mits
```

```
45      0.7071  0.05890000000000006     -0.005700000000000038   -0.0007000
000000000339
50      0.766   0.053200000000000025    -0.006400000000000072
55      0.8192  0.04679999999999995
60      0.866
```

```
Value at 52 is 0.788003
```

In [41]:

```
# Original "data set" --- 21 random numbers between 0 and 1.
x0 = np.linspace(-1,1,21)
y0 = np.random.random(21)
plt.figure(frameon=False,figsize=(10,5))
plt.plot(x0, y0, 'o', label='Data')

# Array with points in between those of the data set for interpolation.
x = np.linspace(-1,1,101)

# Available options for interp1d
options = ('linear', 'nearest', 'zero', 'slinear', 'quadratic', 'cubic', 10)

for o in options:
    f = interp1d(x0, y0,kind=0)    # interpolation function
    plt.plot(x, f(x), label=o)      # plot of interpolated data

plt.legend(loc='best')

plt.show()
```



17/09/2019

Consider the vapour - liquid equilibrium mole fractiom data below for the binary system of methanol and water at 1 atm.

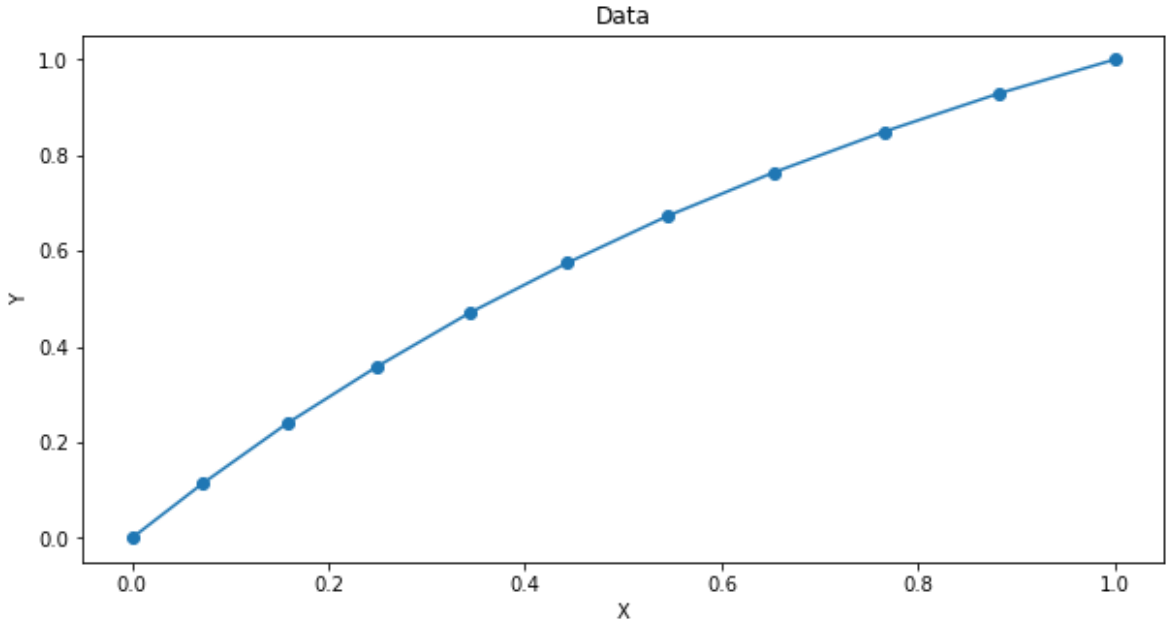
$X = [1, 0.882, 0.765, 0.653, 0.545, 0.443, 0.344, 0.25, 0.159, 0.072, 0]$
$Y = [1, 0.929, 0.849, 0.764, 0.673, 0.575, 0.471, 0.359, 0.241, 0.114, 0]$

Determine the vapour mole fraction of methanol (y) corresponding to the liquid mole fraction of methanol of $x = 0.15$ by linear interpolation and a quadratic Lagrange interpolating

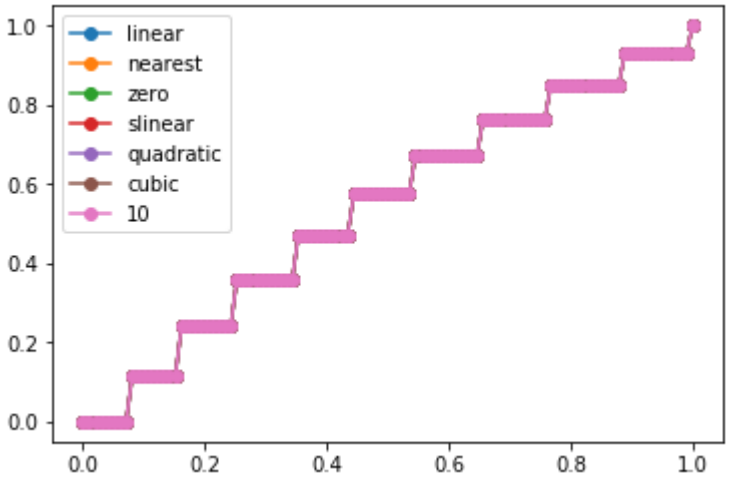
polynomial. Compare your results to the experimental value of $y = 0.517$

In [21]:

```
X = [1,0.882,0.765,0.653,0.545,0.443,0.344,0.25,0.159,0.072,0]
Y = [1,0.929,0.849,0.764,0.673,0.575,0.471,0.359,0.241,0.114,0]

plt.figure(frameon=False,figsize=(10,5))
plt.plot(X, Y,marker = "o")
plt.title("Data")
plt.xlabel("X")
plt.ylabel("Y")
plt.show()

xi = interp1d(Y,X,kind = 2)
yi = interp1d(X,Y,kind = 2)

print("Vapour mole fraction of methanol (y) at x = 0.15 is: {0}".format(yi(0.15)))
print("Experimental value at y = 0.517 is: {0}".format(xi(0.517)))

# Array with points in between those of the data set for interpolation.
x = np.linspace(0,1,100)

# Available options for interp1d
options = ('linear', 'nearest', 'zero', 'slinear', 'quadratic', 'cubic', 10)

for o in options:
    f = interp1d(X, Y, kind=0)    # interpolation function
    plt.plot(x, f(x), label=o,marker = "o")      # plot of interpolated data

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




```
Vapour mole fraction of methanol (y) at x = 0.15 is: 0.2285253925303365
Experimental value at y = 0.517 is: 0.38649191954545176
```



The interpolating polynomial has resulted in this graph