04.02 notebook

In [2]:

```
import numpy as np
amat = np.array([[  4, -1,  1],
                 [ -1,  4, -2],
                 [  1, -2,  4]])

bvec = np.array([ 12, -1,  5])
```

In [29]:

```
# Iteration methon

xini = np.array([0,0,0])
x1 = xini[0]
x2 = xini[1]
x3 = xini[2]

x1new = (  x2 -   x3 + 12)/4
x2new = (  x1 + 2*x3 -  1)/4
x3new = ( -x1 + 2*x2 +  5)/4

print('x1 x2 x3  ', x1new, x2new, x3new)
```

```
x1 x2 x3   3.0 -0.25 1.25
```

In [30]:

```
weight = 0.5

x1 = weight*x1new
x2 = weight*x2new
x3 = weight*x3new

x1new = (  x2 -   x3 + 12)/4
x2new = (  x1 + 2*x3 -  1)/4
x3new = ( -x1 + 2*x2 +  5)/4

print('x1 x2 x3  ', x1new, x2new, x3new)
```

```
x1 x2 x3   2.8125 0.4375 0.8125
```

Use weight to change less x value

In [31]:

```
for i in range(0,50):

    weight = 0.1

    x1p = weight*x1new + (1 - weight)*x1
    x2p = weight*x2new + (1 - weight)*x2
    x3p = weight*x3new + (1 - weight)*x3

    x1 = x1new
    x2 = x2new
    x3 = x3new

    x1new = (  x2p -   x3p + 12)/4
    x2new = (  x1p + 2*x3p -  1)/4
    x3new = ( -x1p + 2*x2p +  5)/4

print('x1 x2 x3  ', x1new, x2new, x3new)
```

```
x1 x2 x3   2.999999999682271 1.0000000004339076 0.9999999995658564
```

In [37]:

```
# Try

A = np.array([[ 2,-1, 0, 1],
              [-1, 2,-1, 0],
              [ 0,-1, 2,-1],
              [ 1, 0,-1, 2]])

B = np.array([0,0,0,1])

np.linalg.solve(A,B)
```

Out [37]: `array([-0.5,  0. ,  0.5,  1. ])`

In [41]:

```
# initial

xini = np.zeros(4)

x1   = xini[0]
x2   = xini[1]
x3   = xini[2]
x4   = xini[3]

x1new = ( x2 - x3 )/2
x2new = ( x1 + x3 )/2
x3new = ( x2 + x4 )/2
x4new = (-x1 + x3 +1)/2

```

```
print('x1 x2 x3 x4  ',x1new, x2new, x3new, x4new)
```

```
array([-0.5,  0. ,  0.5,  1. ])
```

In [42]:

```
for i in range(0,300):

    weight = 0.1

    x1p = weight * x1new + (1 - weight) * x1
    x2p = weight * x2new + (1 - weight) * x2
    x3p = weight * x3new + (1 - weight) * x3
    x4p = weight * x4new + (1 - weight) * x4

    x1 = x1new
    x2 = x2new
    x3 = x3new
    x4 = x4new

    x1new = ( x2p - x4p )/2
    x2new = ( x1p + x3p )/2
    x3new = ( x2p + x4p )/2
    x4new = (-x1p + x3p +1)/2


print('x1 x2 x3 x4  ',x1new, x2new, x3new, x4new)
```

```
x1 x2 x3 x4   -0.5 0.0 0.5 1.0
```

04.03 notebook

In [1]:

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

In [2]:

```
# Lagrange's Method

# Line (n = 2)

# x = 0  2
# y = 7 11

x0 = 0
x1 = 2

y0 = 7
y1 = 11

x  = 1
l0 = ((x - x1) / (x0 - x1))
l1 = ((x - x0) / (x1 - x0))

y = y0*l0 + y1*l1

y
```

Out [2]: 9.0

In [3]:

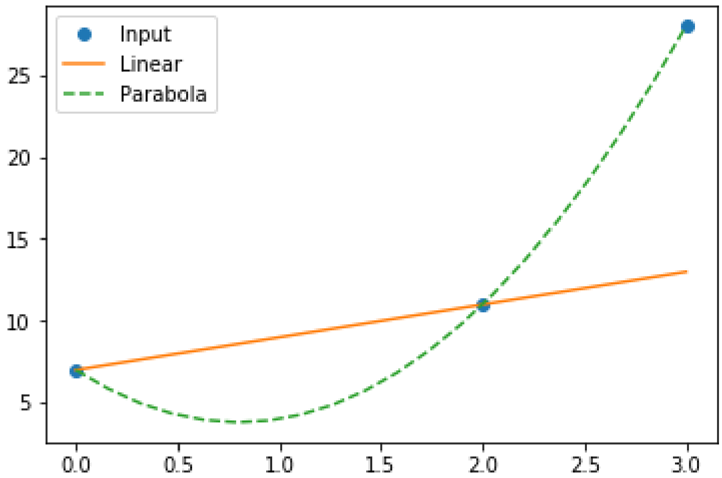
```
# Parabolic (n = 3)

# x = 0  2  3
# y = 7 11  28

xData = np.array([ 0., 2., 3.])
yData = np.array([ 7.,11.,28.])

n = 3
l = []

x = 1.
for i in range(0,n):
    a = 1.
    for j in range(0,n):
        if (i != j):
            a = a * (x - xData[j])/(xData[i] - xData[j])
```

```
    l.append(a)

y = 0
for i in range(0,n):
    y += yData[i]*l[i]

y
```

Out [3]: `4.0`

In [4]:

```
# Ploting graph

x = np.linspace(0,3,20)

# Linear
l0 = ((x - x1) / (x0 - x1))
l1 = ((x - x0) / (x1 - x0))

# Parabolic
l = []
for i in range(0,n):
    a = 1.
    for j in range(0,n):
        if (i != j):
            a = a * (x - xData[j])/(xData[i] - xData[j])
    l.append(a)

fit1 = y0*l0 + y1*l1
fit2 = 0
for i in range(0,n):
    fit2 += yData[i]*l[i]

plt.plot(xData,yData,'o',x,fit1,'-',x,fit2,'--')
plt.legend(['Input','Linear','Parabola'])
plt.show()
```

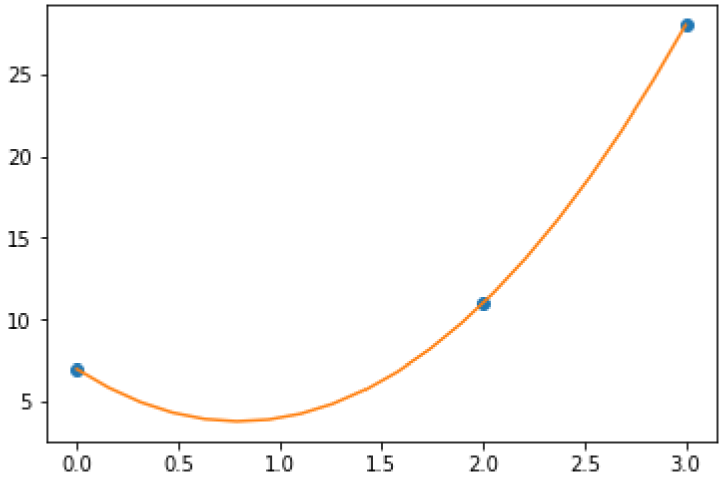


In [5]:

```
def lagrangePoly(x,xData,yDaya):
    n = len(xData)
    l = []
    for i in range(0,n):
        a = 1.
        for j in range(0,n):
            if (i != j):
                a = a * (x - xData[j])/(xData[i] - xData[j])
        l.append(a)

     p = 0
     for i in range(0,n):
         p += yData[i]*l[i]

     return p
```

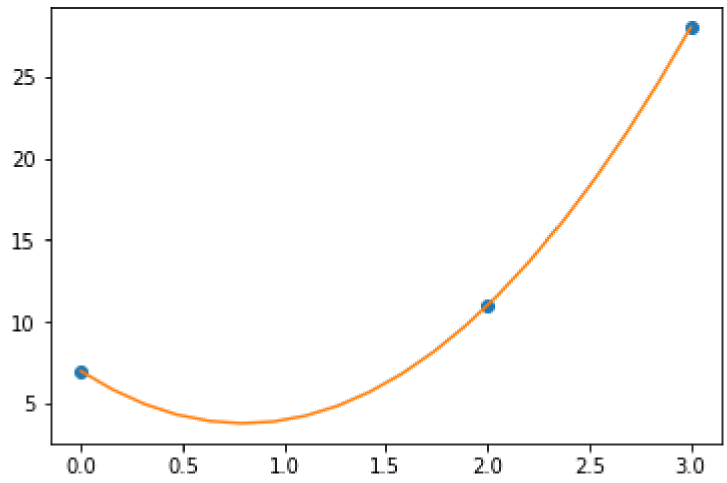
In [6]:

```
xData = np.array([ 0., 2., 3.])
yData = np.array([ 7.,11.,28.])
x = np.linspace(0,3,20)

plt.plot(xData,yData,'o',x,lagrangePoly(x,xData,yData),'-')
plt.show()
```



In [8]:

```
# Newton's Method

# x = 0  2  3
# y = 7 11 28

xData = np.array([ 0., 2., 3.])
yData = np.array([ 7.,11.,28.])

x = np.linspace(0,3,20)

 n = len(xData)

 dy = np.zeros((n,n))

 dy[0] = yData

 for i in range(1,n):
     for j in range(i,n):
         dy[i][j] = (dy[i-1][j] - dy[i-1][i-1])/(xData[j] - xData[i-1])

 a = []
 for i in range(0,n):
     a.append(dy[i][i])

 p = a[n-1]
 for i in range(1,n):
     p = a[(n-1)-i] + (x - xData[(n-1)-i])*p

 plt.plot(xData,yData,'o',x,p,'-')
 plt.show()
```

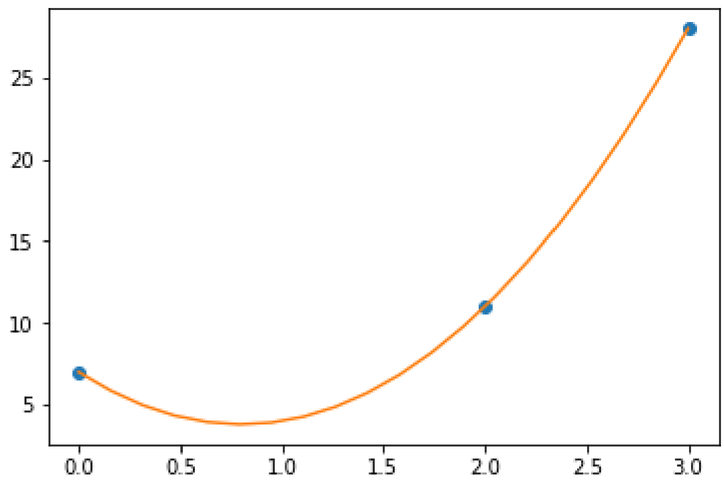


In [9]:

```
def newtonPoly(x,xData,yData):
    # Initiating
    n = len(xData)
    dy = np.zeros((n,n))  # dy is matrix of divided differences

    # Make divided differences
    dy[0] = yData
    for i in range(1,n):
        for j in range(i,n):
            dy[i][j] = (dy[i-1][j] - dy[i-1][i-1])/(xData[j] - xData[i-1])

    a = []
    for i in range(0,n):
        a.append(dy[i][i])

    p = a[n-1]
    for i in range(1,n):
        p = a[(n-1)-i] + (x - xData[(n-1)-i])*p

    return p
```

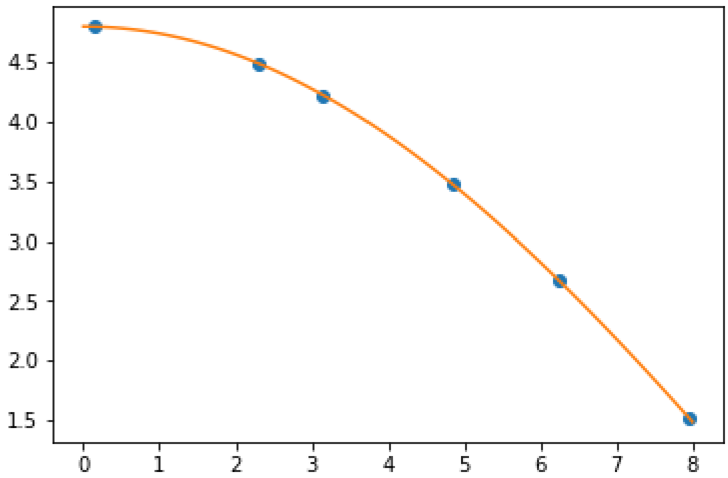
In [10]:

```
xData = np.array([ 0., 2., 3.])
yData = np.array([ 7.,11.,28.])
x = np.linspace(0,3,20)

plt.plot(xData,yData,'o',x,newtonPoly(x,xData,yData),'-')
plt.show()
```



In [11]:

```
# x 0.15    2.30    3.15    4.85    6.25    7.95
# y 4.79867 4.49013 4.2243 3.47313 2.66674 1.51909

xData = np.array([0.15,2.30,3.15,4.85,6.25,7.95])
yData = np.array([4.79867,4.49013,4.2243,3.47313,2.66674,1.51909])
x = np.linspace(0,8,80)

plt.plot(xData,yData,'o',x,newtonPoly(x,xData,yData),'-')
plt.show()
```



In [45]:

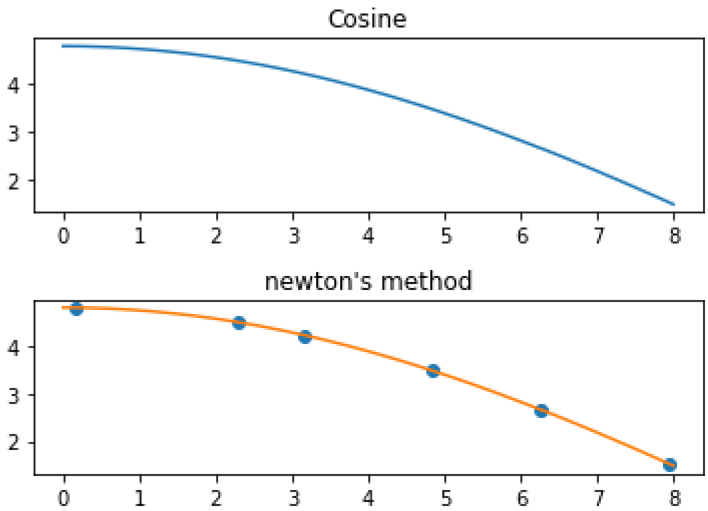
```
import math

x = np.linspace(0,8,17)
y = 4.8*np.cos(np.pi*x/20)

xData = np.array([0.15,2.30,3.15,4.85,6.25,7.95])
yData = np.array([4.79867,4.49013,4.2243,3.47313,2.66674,1.51909])

error_ = newtonPoly(x,xData,yData)-y
error = math.sqrt(np.dot(error_,error_))

print('',y,'\n\n',newtonPoly(x,xData,yData),'\n\n',error_,'\n\nerror :',error)
```

```
[4.8        4.7852032  4.74090403 4.66737562 4.56507128 4.43462176

4.27683132 4.09267279 3.88328157 3.64994863 3.39411255 3.11735063
2.82136921 2.50799311 2.1791544  1.83688048 1.48328157]

[4.80002509 4.78517849 4.74087697 4.6673607  4.56506686 4.43462106
4.27682865 4.09266615 3.88327258 3.64994085 3.39410914 3.11735225
2.82137301 2.50799358 2.17914691 1.83686805 1.48328554]
```

```
[ 2.50944796e-05 -2.47104204e-05 -2.70632809e-05 -1.49197831e-05
 -4.41509820e-06 -6.97032760e-07 -2.66536185e-06 -6.64132739e-06
 -8.99787160e-06 -7.78758468e-06 -3.41152443e-06  1.62202066e-06
  3.79414278e-06  4.67344762e-07 -7.49187110e-06 -1.24292195e-05
  3.96890594e-06]

error : 5.160820292041476e-05
```

In [46]:

```python
x = np.linspace(0,8,85)
y = 4.8*np.cos(np.pi*x/20)

xData = np.array([0.15,2.30,3.15,4.85,6.25,7.95])
yData = np.array([4.79867,4.49013,4.2243,3.47313,2.66674,1.51909])

plt.subplots_adjust(hspace = 0.5)

plt.subplot(2,1,1)
 plt.title('Cosine')
 plt.plot(x,y,'-')

 plt.subplot(2,1,2)
 plt.title('newton\'s method')
 plt.plot(xData,yData,'o',x,newtonPoly(x,xData,yData),'-')
 plt.show()
```



1. Use the Gauss-Seidel method up to five iterative loops to solve the following problem starting with **x = 0**

a) Using a pedestrian implementation in a way you feel comfortable.

```python
## problem 1.a

def gauss_seidel( amat, bmat, x, weight = 1, tol = 1.0e-9 ):
    n = len(x)
    triger = True
    i = 0
    while triger:
        x_ = x.copy()
        for i in range(0,n):
            sum = 0
            for j in range(0,n):
                if (i!=j):
                    sum += amat[i][j]*x_[j]
            x[i] = weight * (bvec[i] - sum) / amat[i][i] + (1 - weight) * x_[i]
        i += 1
        triger = tol < (math.sqrt(np.dot(x-x_,x-x_)))
    return x


amat = np.array([[ 3, 0,-1],
                 [ 0, 4,-2],
                 [-1,-2, 5]])
bvec = np.array([4,10,-10])

x = np.zeros(3)
gauss_seidel( amat, bvec, x )
```

Out: `array([ 1.,  2., -1.])`

b) Using the subroutines in piazza by modifying the segment between '… Modify below …' and '… Up to here …'. (See piazza for sample code).

```python
## problem1.b

import numpy as np
import math

def iterEqs(x,omega): # Omega value supplied by gaussSeidel
    n = len(x)
# ... Modify below ...
    x[0] = omega*(4 + x[2])/3.0 + (1.0 - omega)*x[0]
    x[1] = omega*(10 + 2.0*x[2])/4.0 + (1.0 - omega)*x[1]
    x[2] = omega*(-10 + x[0] + 2.0 * x[1])/5.0 + (1.0 - omega)*x[2]
# ... Up to here ...
    return x

def gaussSeidel(iterEqs,x,tol = 1.0e-9):
    omega = 1.0
    k = 10
    p=1
    for i in range(1,501):
        xOld = x.copy()
        x = iterEqs(x,omega)
        dx = math.sqrt(np.dot(x-xOld,x-xOld))
        if dx < tol: return x,i,omega
# Compute relaxation factor after k+p iterations
        if i == k: dx1 = dx
        if i == k + p:
            dx2 = dx
            omega = 2.0/(1.0 + math.sqrt(1.0  \
                    - (dx2/dx1)**(1.0/p)))
    print('Gauss-Seidel failed to converge')

x = np.zeros(3)
x, numit, omega = gaussSeidel(iterEqs,x)
print("\nNumber of iterations =",numit)
print("\nRelaxation factor =",omega)
print("\nThe solution is:\n",x)
```

Out:

```
Number of iterations = 16

Relaxation factor = 1.0773837106701587

The solution is:
 [ 1.  2. -1.]
```

2. Use the conjugate gradient method to solve the same linear system as in problem 1:
a) Using a pedestrian implementation in a way you feel comfortable.

```python
## problem2.a

import numpy as np
import math

def conjGrad( A, B, x, tol = 1.0e-9 ):
    r = B - np.matmul(A,x)
    s = r.copy()

    i = 0
    while True:
        i += 1
        alpha = np.dot(s,r)/np.dot(s,(np.dot(A,s)))
        x += alpha*s
        r = B - np.matmul(A,x)
        if (math.sqrt(np.dot(r,r)) < tol):
            return x, i
        else:
            beta = -1*np.dot(r,np.dot(A,r))/np.dot(s,(np.dot(A,s)))
            s = r + beta*s

amat = np.array([[ 3,  0,-1],
                 [ 0,  4,-2],
                 [-1,-2,  5]])
bvec = np.array([4,10,-10])

x = np.zeros(3)
conjGrad( amat, bvec, x)
```

Out: `(array([ 1.,  2., -1.]), 23)`

b) Write an interface to the code below (posted on piazza) to supply Av(x) = A*x or by modifying the segment between '… Modify below …' and '… Up to here …'. (See piazza for sample code)

```python
## problem2.b

def Ax(v):    # User supplied function that calculates A*v
#   n = len(v)
    Ax = np.zeros(3)
# ... Modify below ...
    Ax[0] = 3.0*v[0] - v[2]
    Ax[1] = 4.0*v[1] - 2.0*v[2]
    Ax[2] = -v[0] - 2.0*v[1] + 5.0*v[2]
# ... up to here ...
    return Ax

import numpy as np
import math
def conjGrad(Av,x,b,tol=1.0e-9):
    n = len(b)
    r = b - Av(x)
    s = r.copy()
    for i in range(n):
        u = Av(s)
        alpha = np.dot(s,r)/np.dot(s,u)
        x = x + alpha*s
        r = b - Av(x)
        if(math.sqrt(np.dot(r,r))) < tol:
            break
        else:
            beta = -np.dot(r,u)/np.dot(s,u)
            s = r + beta*s
    return x,i



# ... Set problem dimension here ...
n=3
b = np.array([4,10,-10])
x = np.zeros(n)
x,numIter = conjGrad(Ax,x,b)
print("\nThe solution is:\n",x)
print("\nNumber of iterations =",numIter)
```

Out:

```
The solution is:
 [ 1.  2. -1.]

Number of iterations = 2
```