04.02 notebook

In [2]:

1. import numpy as np
2. amat = np.array([[ 4, -1, 1],
3. [ -1, 4, -2],
4. [ 1, -2, 4]])
6. bvec = np.array([ 12, -1, 5])

In [29]:

1. *# Iteration methon*
3. xini = np.array([0,0,0])
4. x1 = xini[0]
5. x2 = xini[1]
6. x3 = xini[2]
8. x1new = ( x2 - x3 + 12)/4
9. x2new = ( x1 + 2\*x3 - 1)/4
10. x3new = ( -x1 + 2\*x2 + 5)/4
12. print('x1 x2 x3 ', x1new, x2new, x3new)

x1 x2 x3 3.0 -0.25 1.25

In [30]:

1. weight = 0.5
3. x1 = weight\*x1new
4. x2 = weight\*x2new
5. x3 = weight\*x3new
7. x1new = ( x2 - x3 + 12)/4
8. x2new = ( x1 + 2\*x3 - 1)/4
9. x3new = ( -x1 + 2\*x2 + 5)/4
11. 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]:

1. for i in range(0,50):
3. weight = 0.1
5. x1p = weight\*x1new + (1 - weight)\*x1
6. x2p = weight\*x2new + (1 - weight)\*x2
7. x3p = weight\*x3new + (1 - weight)\*x3
9. x1 = x1new
10. x2 = x2new
11. x3 = x3new
13. x1new = ( x2p - x3p + 12)/4
14. x2new = ( x1p + 2\*x3p - 1)/4
15. x3new = ( -x1p + 2\*x2p + 5)/4
17. print('x1 x2 x3 ', x1new, x2new, x3new)

 x1 x2 x3 2.999999999682271 1.0000000004339076 0.9999999995658564

In [37]:

1. *# Try*
3. A = np.array([[ 2,-1, 0, 1],
4. [-1, 2,-1, 0],
5. [ 0,-1, 2,-1],
6. [ 1, 0,-1, 2]])
8. B = np.array([0,0,0,1])
10. np.linalg.solve(A,B)

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

In [41]:

1. *# initial*
3. xini = np.zeros(4)
5. x1 = xini[0]
6. x2 = xini[1]
7. x3 = xini[2]
8. x4 = xini[3]
10. x1new = ( x2 - x3 )/2
11. x2new = ( x1 + x3 )/2
12. x3new = ( x2 + x4 )/2
13. x4new = (-x1 + x3 +1)/2
15. print('x1 x2 x3 x4 ',x1new, x2new, x3new, x4new)

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

In [42]:

1. for i in range(0,300):
3. weight = 0.1
5. x1p = weight \* x1new + (1 - weight) \* x1
6. x2p = weight \* x2new + (1 - weight) \* x2
7. x3p = weight \* x3new + (1 - weight) \* x3
8. x4p = weight \* x4new + (1 - weight) \* x4
10. x1 = x1new
11. x2 = x2new
12. x3 = x3new
13. x4 = x4new
15. x1new = ( x2p - x4p )/2
16. x2new = ( x1p + x3p )/2
17. x3new = ( x2p + x4p )/2
18. x4new = (-x1p + x3p +1)/2

21. 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]:

1. import numpy as np
2. import matplotlib.pyplot as plt

In [2]:

1. *# Lagrange's Method*
3. *# Line (n = 2)*
5. *# x = 0 2*
6. *# y = 7 11*
8. x0 = 0
9. x1 = 2
11. y0 = 7
12. y1 = 11
14. x = 1
15. l0 = ((x - x1) / (x0 - x1))
16. l1 = ((x - x0) / (x1 - x0))
18. y = y0\*l0 + y1\*l1
20. y

Out [2]: 9.0

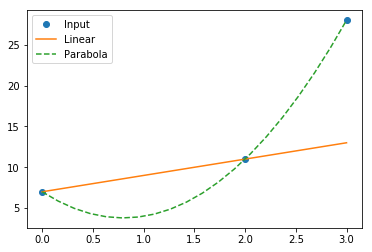
In [3]:

1. *# Parabolic (n = 3)*
3. *# x = 0 2 3*
4. *# y = 7 11 28*
6. xData = np.array([ 0., 2., 3.])
7. yData = np.array([ 7.,11.,28.])
9. n = 3
10. l = []
12. x = 1.
13. for i in range(0,n):
14. a = 1.
15. for j in range(0,n):
16. if (i != j):
17. a = a \* (x - xData[j])/(xData[i] - xData[j])
18. l.append(a)
20. y = 0
21. for i in range(0,n):
22. y += yData[i]\*l[i]
24. y

Out [3]: 4.0

In [4]:

1. *# Ploting graph*
3. x = np.linspace(0,3,20)
5. *# Linear*
6. l0 = ((x - x1) / (x0 - x1))
7. l1 = ((x - x0) / (x1 - x0))
9. *# Parabolic*
10. l = []
11. for i in range(0,n):
12. a = 1.
13. for j in range(0,n):
14. if (i != j):
15. a = a \* (x - xData[j])/(xData[i] - xData[j])
16. l.append(a)
18. fit1 = y0\*l0 + y1\*l1
19. fit2 = 0
20. for i in range(0,n):
21. fit2 += yData[i]\*l[i]
23. plt.plot(xData,yData,'o',x,fit1,'-',x,fit2,'--')
24. plt.legend(['Input','Linear','Parabola'])
25. plt.show()

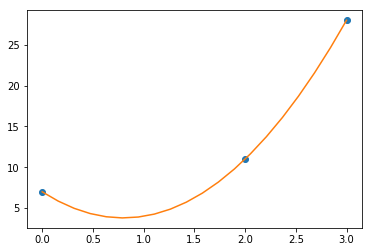


In [5]:

1. def lagrangePoly(x,xData,yDaya):
2. n = len(xData)
3. l = []
4. for i in range(0,n):
5. a = 1.
6. for j in range(0,n):
7. if (i != j):
8. a = a \* (x - xData[j])/(xData[i] - xData[j])
9. l.append(a)
11. p = 0
12. for i in range(0,n):
13. p += yData[i]\*l[i]
15. return p

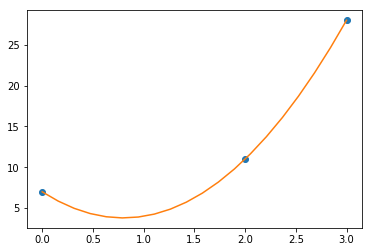
In [6]:

1. xData = np.array([ 0., 2., 3.])
2. yData = np.array([ 7.,11.,28.])
3. x = np.linspace(0,3,20)
5. plt.plot(xData,yData,'o',x,lagrangePoly(x,xData,yData),'-')
6. plt.show()



In [8]:

1. *# Newton's Method*
3. *# x = 0 2 3*
4. *# y = 7 11 28*
6. xData = np.array([ 0., 2., 3.])
7. yData = np.array([ 7.,11.,28.])
9. x = np.linspace(0,3,20)
11. n = len(xData)
13. dy = np.zeros((n,n))
15. dy[0] = yData
17. for i in range(1,n):
18. for j in range(i,n):
19. dy[i][j] = (dy[i-1][j] - dy[i-1][i-1])/(xData[j] - xData[i-1])
21. a = []
22. for i in range(0,n):
23. a.append(dy[i][i])
25. p = a[n-1]
26. for i in range(1,n):
27. p = a[(n-1)-i] + (x - xData[(n-1)-i])\*p
29. plt.plot(xData,yData,'o',x,p,'-')
30. plt.show()

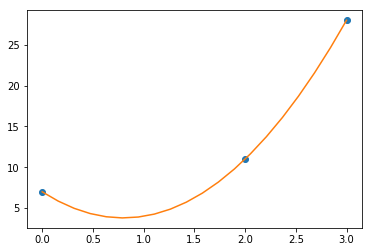


In [9]:

1. def newtonPoly(x,xData,yData):
2. *# Initiating*
3. n = len(xData)
4. dy = np.zeros((n,n)) *# dy is matrix of divided differences*
6. *# Make divided differences*
7. dy[0] = yData
8. for i in range(1,n):
9. for j in range(i,n):
10. dy[i][j] = (dy[i-1][j] - dy[i-1][i-1])/(xData[j] - xData[i-1])
12. a = []
13. for i in range(0,n):
14. a.append(dy[i][i])
16. p = a[n-1]
17. for i in range(1,n):
18. p = a[(n-1)-i] + (x - xData[(n-1)-i])\*p
20. return p

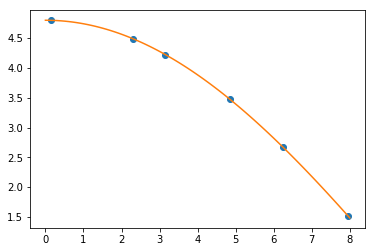
In [10]:

1. xData = np.array([ 0., 2., 3.])
2. yData = np.array([ 7.,11.,28.])
3. x = np.linspace(0,3,20)
5. plt.plot(xData,yData,'o',x,newtonPoly(x,xData,yData),'-')
6. plt.show()



In [11]:

1. *# x 0.15 2.30 3.15 4.85 6.25 7.95*
2. *# y 4.79867 4.49013 4.2243 3.47313 2.66674 1.51909*
4. xData = np.array([0.15,2.30,3.15,4.85,6.25,7.95])
5. yData = np.array([4.79867,4.49013,4.2243,3.47313,2.66674,1.51909])
6. x = np.linspace(0,8,80)
8. plt.plot(xData,yData,'o',x,newtonPoly(x,xData,yData),'-')
9. plt.show()



In [45]:

1. import math
3. x = np.linspace(0,8,17)
4. y = 4.8\*np.cos(np.pi\*x/20)
6. xData = np.array([0.15,2.30,3.15,4.85,6.25,7.95])
7. yData = np.array([4.79867,4.49013,4.2243,3.47313,2.66674,1.51909])
9. error\_ = newtonPoly(x,xData,yData)-y
10. error = math.sqrt(np.dot(error\_,error\_))
12. 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

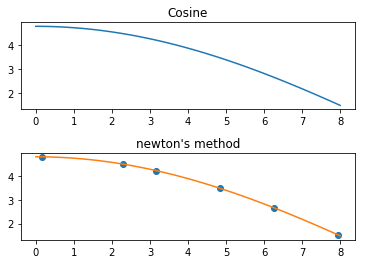
3.79414278e-06 4.67344762e-07 -7.49187110e-06 -1.24292195e-05

3.96890594e-06]

error : 5.160820292041476e-05

In [46]:

1. x = np.linspace(0,8,85)
2. y = 4.8\*np.cos(np.pi\*x/20)
4. xData = np.array([0.15,2.30,3.15,4.85,6.25,7.95])
5. yData = np.array([4.79867,4.49013,4.2243,3.47313,2.66674,1.51909])
7. plt.subplots\_adjust(hspace = 0.5)
9. plt.subplot(2,1,1)
10. plt.title('Cosine')
11. plt.plot(x,y,'-')
13. plt.subplot(2,1,2)
14. plt.title('newton\'s method')
15. plt.plot(xData,yData,'o',x,newtonPoly(x,xData,yData),'-')
16. 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.

1. *## problem 1.a*
3. def gauss\_seidel( amat, bmat, x, weight = 1, tol = 1.0e-9 ):
4. n = len(x)
5. triger = True
6. i = 0
7. while triger:
8. x\_ = x.copy()
9. for i in range(0,n):
10. sum = 0
11. for j in range(0,n):
12. if (i!=j):
13. sum += amat[i][j]\*x\_[j]
14. x[i] = weight \* (bvec[i] - sum) / amat[i][i] + (1 - weight) \* x\_[i]
15. i += 1
16. triger = tol < (math.sqrt(np.dot(x-x\_,x-x\_)))
17. return x

20. amat = np.array([[ 3, 0,-1],
21. [ 0, 4,-2],
22. [-1,-2, 5]])
23. bvec = np.array([4,10,-10])
25. x = np.zeros(3)
26. 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).

1. *## problem1.b*
3. import numpy as np
4. import math
6. def iterEqs(x,omega): *# Omega value supplied by gaussSeidel*
7. n = len(x)
8. *# ... Modify below ...*
9. x[0] = omega\*(4 + x[2])/3.0 + (1.0 - omega)\*x[0]
10. x[1] = omega\*(10 + 2.0\*x[2])/4.0 + (1.0 - omega)\*x[1]
11. x[2] = omega\*(-10 + x[0] + 2.0 \* x[1])/5.0 + (1.0 - omega)\*x[2]
12. *# ... Up to here ...*
13. return x
15. def gaussSeidel(iterEqs,x,tol = 1.0e-9):
16. omega = 1.0
17. k = 10
18. p=1
19. for i in range(1,501):
20. xOld = x.copy()
21. x = iterEqs(x,omega)
22. dx = math.sqrt(np.dot(x-xOld,x-xOld))
23. if dx < tol: return x,i,omega
24. *# Compute relaxation factor after k+p iterations*
25. if i == k: dx1 = dx
26. if i == k + p:
27. dx2 = dx
28. omega = 2.0/(1.0 + math.sqrt(1.0 \
29. - (dx2/dx1)\*\*(1.0/p)))
30. print('Gauss-Seidel failed to converge')
32. x = np.zeros(3)
33. x, numit, omega = gaussSeidel(iterEqs,x)
34. print("\nNumber of iterations =",numit)
35. print("\nRelaxation factor =",omega)
36. 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.

1. *## problem2.a*
3. import numpy as np
4. import math
6. def conjGrad( A, B, x, tol = 1.0e-9 ):
7. r = B - np.matmul(A,x)
8. s = r.copy()
10. i = 0
11. while True:
12. i += 1
13. alpha = np.dot(s,r)/np.dot(s,(np.dot(A,s)))
14. x += alpha\*s
15. r = B - np.matmul(A,x)
16. if (math.sqrt(np.dot(r,r)) < tol):
17. return x, i
18. else:
19. beta = -1\*np.dot(r,np.dot(A,r))/np.dot(s,(np.dot(A,s)))
20. s = r + beta\*s
22. amat = np.array([[ 3, 0,-1],
23. [ 0, 4,-2],
24. [-1,-2, 5]])
25. bvec = np.array([4,10,-10])
27. x = np.zeros(3)
28. 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)

1. *## problem2.b*
3. def Ax(v): *# User supplied function that calculates A\*v*
4. *# n = len(v)*
5. Ax = np.zeros(3)
6. *# ... Modify below ...*
7. Ax[0] = 3.0\*v[0] - v[2]
8. Ax[1] = 4.0\*v[1] - 2.0\*v[2]
9. Ax[2] = -v[0] - 2.0\*v[1] + 5.0\*v[2]
10. *# ... up to here ...*
11. return Ax
13. import numpy as np
14. import math
15. def conjGrad(Av,x,b,tol=1.0e-9):
16. n = len(b)
17. r = b - Av(x)
18. s = r.copy()
19. for i in range(n):
20. u = Av(s)
21. alpha = np.dot(s,r)/np.dot(s,u)
22. x = x + alpha\*s
23. r = b - Av(x)
24. if(math.sqrt(np.dot(r,r))) < tol:
25. break
26. else:
27. beta = -np.dot(r,u)/np.dot(s,u)
28. s = r + beta\*s
29. return x,i


33. *# ... Set problem dimension here ...*
34. n=3
35. b = np.array([4,10,-10])
36. x = np.zeros(n)
37. x,numIter = conjGrad(Ax,x,b)
38. print("\nThe solution is:\n",x)
39. print("\nNumber of iterations =",numIter)

Out:

The solution is:

[ 1. 2. -1.]

Number of iterations = 2