In [19]:

```
#let us use numpy library
import numpy as np

x = np.array([1, 2, 3, 6, 9, 13]) #define a numpy array

print( x )
print( x[0] ) #slicing
print( x[1:-2] ) #slicing
```

```
[ 1  2  3  6  9 13]
1
[2 3 6]
```

Numpy operations and slicing

In [21]:

```
import math
y1_list = [2*x_i for x_i in x_list]
y2_list = [x_i**2 + 2*x_i + 1 for x_i in x_list]
y3_list = [math.log10(x_i) for x_i in x_list]
y4_list = [x_i > 4 for x_i in x_list]

print( 'x_list = ', x_list )
print( 'y1_list = ', y1_list )
print( 'y2_list = ',  y2_list )
print( 'y3_list = ',  y3_list, '\n and rounded y3_list = ',  [round(y3_i, 4) for
 y3_i in y3_list] )
print( 'y4_list = ',  y4_list )
```

```
x_list =  [1, 2, 3, 6, 9, 13]
y1_list =  [2, 4, 6, 12, 18, 26]
y2_list =  [4, 9, 16, 49, 100, 196]
y3_list =  [0.0, 0.3010299956639812, 0.47712125471966244, 0.77815125
03836436, 0.9542425094393249, 1.1139433523068367]
 and rounded y3_list =  [0.0, 0.301, 0.4771, 0.7782, 0.9542, 1.1139]
y4_list =  [False, False, False, True, True, True]
```

In [ ]:

```
#let us use numpy library
y1 = 2*x
y2 = x**2 + 2*x + 1
y3 = np.log10(x)
y4 = x > 4
print( 'x = ', x )
print( 'y1 = ', y1 )
print( 'y2 = ',  y2 )
print('y3 = ',  y3, '\n and rounded y3 = ',  np.round(y3, 4) )
print( 'y4 = ',  y4 )
```

Generating a numpy sequence

In [ ]:

```
x = [i for i in range(0, 10, 2)] #arguments are (start, stop, step); by default
 start is zero and step is 1
print( x )
#Otherwise, list(range(0, 10, 2))
```

In [ ]:

```
#let us use numpy library
x = np.arange(0, 10, 2) #arguments are (start, stop, step); by default start is
 zero and step is 1
print( x )
```

Exercise 1.3.1

Use numpy to determine the roots of $x^2 + bx + 1 = 0$ for $b \in (4, 11] \cap \mathbb{Z}$

Hint: $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ are roots of $ax^2 + bx + c = 0$

In [ ]:

```
#answer
a = 1
b = np.arange(5, 12)
c = 1
delta = np.sqrt(b**2 - 4*a*c)
print('b = ', b)
print( 'with +:', np.round((-b-delta)/(2*a), 2), 'or')
print( 'with -:', np.round((-b+delta)/(2*a), 2) )
```

Exercise 1.3.2

Plot $sin(2\pi ft)exp(-t/2)$ for t=0 to t=10 with a step size of 1/50. Use numpy to define arrays and make calculations.

Hint: If python lists were used as in section 8 of tutorial 1,

In [ ]:

```
import math
import matplotlib.pyplot as pl
%matplotlib inline

t = [i/50 for i in range(501)] #generate values from 0 to 10
f = 1
y = [math.sin(2*math.pi*f*i)*math.exp(-i/2) for i in t] # y = sin(2Pi*f*t)*exp(-
t/2)
pl.scatter(t,y)
pl.show()
```

In [ ]:

```
#answer
import matplotlib.pyplot as pl
%matplotlib inline

t = np.arange(501)/50
f = 1
y = np.sin(2*np.pi*f*t) * np.exp(-t/2)
pl.scatter(t, y)
pl.show()
```

In [ ]:

```
#Try the following code in a non-ipyhon environment
"""
from matplotlib.widgets import Slider

ax = pl.subplot(111)
pl.subplots_adjust(left=0.25, bottom=0.25)
t = np.arange(0.0, 10, 0.02)
a0 = 0
f0 = 1
s = np.exp(a0*t)*np.sin(2*np.pi*f0*t)
l, = pl.plot(t,s, lw=2, color='red')

axfreq = pl.axes([0.25, 0.1, 0.65, 0.03])
axdamp  = pl.axes([0.25, 0.15, 0.65, 0.03])

sfreq = Slider(axfreq, 'Freq', 0.5, 3.0, valinit=f0)
sdamp = Slider(axdamp, 'Damp', -0.5, 0.5, valinit=a0)

def update(val):
    damp = sdamp.val
    freq = sfreq.val
    l.set_ydata(np.exp(damp*t)*np.sin(2*np.pi*freq*t))
    pl.draw()
sfreq.on_changed(update)
sdamp.on_changed(update)

pl.show()
"""
```

Matrix operations

In [2]:

```
# Create a new array with three elements
import  numpy as np
A = np.array([1, 2, 3])
print( 'A = \n {}'.format(A) )
print( 'shape of A is {} \n'.format(A.shape) )

B = np.array([[1, 2, 3]])
print( 'B = \n {}'.format(B) )
print( 'shape of B is {} \n'.format(B.shape) )

#Create a 2x3 array
C = np.array([[1, 2, 3],[4, 5, 6]])
print( 'C = \n {}'.format(C) )
print( 'shape of C is {} \n'.format(C.shape) )

#Create a 3x3 array
D = np.array([[1, 2, 3],[3, 5, 6],[2, 8, 11]])
print( 'D = \n {}'.format(D) )
print( 'shape of D is {} \n'.format(D.shape) )

#Create a 3x2 array
E = np.array([[2, 0],[4, 1], [1, 2]])
print( 'E = \n {}'.format(E) )
print( 'shape of E is {} \n'.format(E.shape) )

F = np.array([[4, 2, 11, 3]])
print( 'F = \n {}'.format(F) )
print( 'shape of F is {} \n'.format(F.shape) )
```

```
A = 
 [1 2 3]
shape of A is (3,) 

B = 
 [[1 2 3]]
shape of B is (1, 3) 

C = 
 [[1 2 3]
 [4 5 6]]
shape of C is (2, 3) 

D = 
 [[ 1  2  3]
 [ 3  5  6]
 [ 2  8 11]]
shape of D is (3, 3) 

E = 
 [[2 0]
 [4 1]
 [1 2]]
shape of E is (3, 2) 

F = 
 [[ 4  2 11  3]]
shape of F is (1, 4) 
```

Exercise 1.4.1

Perform the following matrix operations.

1. E'
2. $E' \circ C$ (elementwise matrix multiplication)
3. $E \cdot C$ (matrix multiplication)
4. $B' \cdot F$
5. $A' \cdot F$

Hint: Identify the use of A[np.newaxis, :], A[:, np.newaxis], A[:, np.newaxis, :], etc. Alternative to [:, np.newaxis] is [:, None].

In [ ]:

```
#Answers

Et = E.T
print( 'Et = \n {} \n'.format(Et) )

EtoC = E.T*C
print( 'E\'oC = \n {} \n'.format(EtoC) )

EC = E.dot(C)
print( 'E.C\' = \n {} \n'.format(EC) ) #or np.dot(E, C)

BtF = B.T.dot(F)
print( 'B\'.F = \n {} \n'.format(BtF))

AtF = A[np.newaxis, :].T.dot(F)
print( 'A\'.F = \n {} \n'.format(AtF))
```

Exercise 1.4.2

For random square matrices A and B, show that the following identy is valid.

$$(A^\top B)^{-1} = B^{-1} A^{-\top} \$$$

Hint: The following numpy methods maybe useful; np.random.random((N,N)), np.linalg.inv() and np.allclose().

In [18]:

```
import numpy as np
A=np.random.random((4,4))
B=np.random.random((4,4))
At=A.T
Bt=B.T
Ai=np.linalg.inv(A)
Bi=np.linalg.inv(B)
AtB=A.T.dot(B)
Ati=np.linalg.inv(At)
AtBinv=np.linalg.inv(AtB)
BiAt=np.linalg.inv(B).dot(Ati)

a=np.sqrt(AtBinv.dot(AtBinv))
b=np.sqrt(BiAt.dot(BiAt))


sim_cos=AtBinv.dot(BiAt)/(a*b)
print('sim_cos =',sim_cos)

L=np.sqrt(sim_cos.dot(sim_cos))
print('L=', L)
```

```
sim_cos = [[nan  1. nan nan]
 [nan  1. nan nan]
 [ 1. nan  1.  1.]
 [nan  1. nan nan]]
L= [[nan nan nan nan]
 [nan nan nan nan]
 [nan nan nan nan]
 [nan nan nan nan]]
```

```
/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:13: Run
timeWarning: invalid value encountered in sqrt
  del sys.path[0]
/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:14: Run
timeWarning: invalid value encountered in sqrt
```

Matrix Decomposition

Matrix decomposition using numpy

Exercise 2.1.1

Decompose the following matrix A using,

1. QR decomposition
2. Eigendecomposition

Verify,

1. Eigenvalues obtained from numpy Eigendecomposition are Eigenvalues of A
2. Eigenvector matrix is an orthogonal matrix

In [20]:

```
A = np.diag((1, 2, 3))
print(A)
```

```
[[1 0 0]
 [0 2 0]
 [0 0 3]]
```

In [33]:

```
l, v = np.linalg.eig(A)
print(l, v)
print((A-l*np.eye(A.shape[0])).dot(v)) # (A-lI)U = 0
print(v.T.dot(v)) #orthogonal if U'U = UU'= I or you can use inverse
```

```
---------------------------------------------------------------------
-------
LinAlgError                               Traceback (most recent cal
l last)
<ipython-input-33-1a1fffc0c516> in <module>()
----> 1 l, v = np.linalg.eig(A)
      2 print(l, v)
      3 print((A-l*np.eye(A.shape[0])).dot(v)) # (A-lI)U = 0
      4 print(v.T.dot(v)) #orthogonal if U'U = UU'= I or you can use
 inverse

/anaconda3/lib/python3.6/site-packages/numpy/linalg/linalg.py in eig
(a)
   1140     a, wrap = _makearray(a)
   1141     _assertRankAtLeast2(a)
-> 1142     _assertNdSquareness(a)
   1143     _assertFinite(a)
   1144     t, result_t = _commonType(a)

/anaconda3/lib/python3.6/site-packages/numpy/linalg/linalg.py in _as
sertNdSquareness(*arrays)
    209     for a in arrays:
    210         if max(a.shape[-2:]) != min(a.shape[-2:]):
--> 211             raise LinAlgError('Last 2 dimensions of the arra
y must be square')
    212
    213 def _assertFinite(*arrays):

LinAlgError: Last 2 dimensions of the array must be square
```

In [31]:

```
Q, R = np.linalg.qr(A)
print('Q=\n', np.round(Q, 2))
print('\nR=\n', np.round(R, 2))

#A = QR of an orthogonal matrix Q and an upper triangular matrix R
#A = LU of an lower triangular matrix L and an upper triangular matrix U
```

```
Q=
 [[-0.45 -0.89]
 [-0.89  0.45]]

R=
 [[ -8.94 -11.18  -4.47]
 [  0.    -6.71 -13.42]]
```

Singular value decomposition

A matrix A of size $m \times n$ can be decomposed into,

$$A = U\Sigma V'$$

where

U is a $m \times r$ unitary matrix (i.e for this context, $U^\top U = I$)

Σ is a $r \times r$ diagonal matrix. Diagonal elements are *called singular values* which are non-negative.

V' is a $r \times n$ unitary matrix

In [39]:

```
#print('============SVD===================\n')
#M = np.mat("4 11 14;8 7 -2")
#print('M\n',M)
#U,sigma,Vt = np.linalg.svd( M,full_matrices = False) #这里的V其实是V.H
#print('\nU\n',U)
#print('\nsigma\n',sigma)#注意此处的sigma是一个一维数组,验证的话需要将其转化为对角阵
#Sigma_martix=np.diag(sigma)
#print('\nSigma_martix\n', Sigma_martix)
#print('\nVt\n',Vt)

#M_re=U.dot(Sigma_martix.dot(Vt))
#print('\nvalidation of the svd\n')

#print( 'M_re = {} \n'.format(np.round(M_re, 2)) )
#print( 'Is M close to M_re? ', np.allclose(M, M_re) )
```

In [1]:

```
import numpy as np

#user vs movie, a 7x5 matrix
A = np.array([[1, 1, 1, 0, 0],\
              [3, 3, 3, 0, 0],\
              [4, 4, 4, 0, 0],\
              [5, 5, 5, 0, 0],\
              [0, 2, 0, 4, 4],\
              [0, 0, 0, 5, 5],\
              [0, 1, 0, 2, 2]])

U, s, Vt = np.linalg.svd(A, full_matrices=False)
S = np.diag(s)

print( 'U = {} \n\n s = {} \n\n Vt = {} \n'.format(np.round(U, 2), np.round(S, 2
), np.round(Vt, 2)) )

A_reconstructed = U.dot(S.dot(Vt))

print( 'A_reconstructed = {} \n'.format(np.round(A_reconstructed, 2)) )
print( 'Is A close to A_reconstructed? ', np.allclose(A, A_reconstructed) )
```

```
U = [[-0.14 -0.02 -0.01  0.56 -0.38]
 [-0.41 -0.07 -0.03  0.21  0.76]
 [-0.55 -0.09 -0.04 -0.72 -0.18]
 [-0.69 -0.12 -0.05  0.34 -0.23]
 [-0.15  0.59  0.65  0.    0.2 ]
 [-0.07  0.73 -0.68  0.    0.  ]
 [-0.08  0.3   0.33  0.   -0.4 ]]

 s = [[ 12.48   0.     0.     0.     0.  ]
 [  0.     9.51   0.     0.     0.  ]
 [  0.     0.     1.35   0.     0.  ]
 [  0.     0.     0.     0.     0.  ]
 [  0.     0.     0.     0.     0.  ]]

 Vt = [[-0.56 -0.59 -0.56 -0.09 -0.09]
 [-0.13  0.03 -0.13  0.7   0.7 ]
 [-0.41  0.8  -0.41 -0.09 -0.09]
 [-0.71  0.    0.71  0.    0.  ]
 [ 0.   -0.    0.   -0.71  0.71]]

A_reconstructed = [[ 1.  1.  1. -0. -0.]
 [ 3.  3.  3. -0. -0.]
 [ 4.  4.  4. -0. -0.]
 [ 5.  5.  5. -0. -0.]
 [-0.  2. -0.  4.  4.]
 [-0.  0. -0.  5.  5.]
 [-0.  1. -0.  2.  2.]]

Is A close to A_reconstructed?  True
```

In [45]:

```
#A sanity check
print( np.allclose(U.T.dot(U), np.eye(2)) )
print( np.allclose(Vt.dot(Vt.T), np.eye(2)) )

print( np.sqrt(A.dot(A.T)))
```

```
True
True
[[18.24828759  9.        ]
 [ 9.         10.81665383]]
```

Exercise 2.2.1

1. What is the rank of A? Hint: Perform EROs. Why are some of the singular values close to zero?
2. Calculate the compression ratio.
3. Reconstruct the matrix using only the most significant singular values.

In [2]:

```
#answer
for n_components in range(1,S.shape[0]):
    A_hat_reconstructed = U[0:U.shape[0], 0:n_components]\
       .dot(S[0:n_components,0:n_components])\
       .dot(Vt[0:n_components, 0:Vt.shape[1]])
    SSE = np.sum((A - A_hat_reconstructed)**2)
    approx_ratio = n_components/A.shape[1]
    comp_ratio = (A.shape[1]*n_components + n_components + A.shape[0]*n_componen
ts)/(A.shape[1] * A.shape[0])

    #print(np.round(a,2))
    print('If we choose {} dominant singular value/s;\n A_hat_reconstructed = \n
{} \n SSE = {} and\n compresion ratio = {}\n\n'\
          .format(n_components, np.round(A_hat_reconstructed, 2), SSE, np.round(
comp_ratio,10)))
```

```
If we choose 1 dominant singular value/s;
 A_hat_reconstructed =
[[ 0.97  1.02  0.97  0.15  0.15]
 [ 2.9   3.05  2.9   0.46  0.46]
 [ 3.86  4.07  3.86  0.62  0.62]
 [ 4.83  5.09  4.83  0.77  0.77]
 [ 1.07  1.13  1.07  0.17  0.17]
 [ 0.51  0.53  0.51  0.08  0.08]
 [ 0.54  0.57  0.54  0.09  0.09]]
 SSE = 92.22427221863023 and
 compresion ratio = 0.3714285714


If we choose 2 dominant singular value/s;
 A_hat_reconstructed =
[[ 0.99  1.01  0.99 -0.   -0.  ]
 [ 2.98  3.04  2.98 -0.   -0.  ]
 [ 3.98  4.05  3.98 -0.01 -0.01]
 [ 4.97  5.06  4.97 -0.01 -0.01]
 [ 0.36  1.29  0.36  4.08  4.08]
 [-0.37  0.73 -0.37  4.92  4.92]
 [ 0.18  0.65  0.18  2.04  2.04]]
 SSE = 1.810530940559784 and
 compresion ratio = 0.7428571429


If we choose 3 dominant singular value/s;
 A_hat_reconstructed =
[[ 1.  1.  1. -0. -0.]
 [ 3.  3.  3. -0. -0.]
 [ 4.  4.  4. -0. -0.]
 [ 5.  5.  5. -0. -0.]
 [-0.  2. -0.  4.  4.]
 [-0.  0. -0.  5.  5.]
 [-0.  1. -0.  2.  2.]]
 SSE = 8.767996136259424e-29 and
 compresion ratio = 1.1142857143


If we choose 4 dominant singular value/s;
 A_hat_reconstructed =
[[ 1.  1.  1. -0. -0.]
 [ 3.  3.  3. -0. -0.]
 [ 4.  4.  4. -0. -0.]
 [ 5.  5.  5. -0. -0.]
 [-0.  2. -0.  4.  4.]
 [-0.  0. -0.  5.  5.]
 [-0.  1. -0.  2.  2.]]
 SSE = 8.770461326588239e-29 and
 compresion ratio = 1.4857142857
```

Exercise 2.2.2 Make use of SVD to compress a gray-scale image.

In [48]:

```
import numpy as np
import matplotlib.pyplot as pl
from scipy import misc
%matplotlib inline

A = misc.lena() #or use misc.lena()
```

```
---------------------------------------------------------------------
-------
AttributeError                            Traceback (most recent cal
l last)
<ipython-input-48-17db9601c838> in <module>()
      4 get_ipython().run_line_magic('matplotlib', 'inline')
      5
----> 6 A = misc.lena() #or use misc.lena()

AttributeError: module 'scipy.misc' has no attribute 'lena'
```

In [49]:

```
n_components = 50

U, s, Vt = np.linalg.svd(A, full_matrices=False)
S = np.diag(s)

A_hat_reconstructed = U[0:U.shape[0], 0:n_components]\
       .dot(S[0:n_components,0:n_components])\
       .dot(Vt[0:n_components, 0:Vt.shape[1]])
SSE = np.sum((A - A_hat_reconstructed)**2)
comp_ratio = (A.shape[1]*n_components + n_components + A.shape[0]*n_components)/
(A.shape[1] * A.shape[0])

print('If we choose {} dominant singular value/s;\n SSE = {} and\n compresion ra
tio = {}\n\n'\
      .format(n_components, SSE, np.round(comp_ratio,10)))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-49-5f465681aa42> in <module>()
      5 
      6 A_hat_reconstructed = U[0:U.shape[0], 0:n_components]
.dot(S[0:n_components,0:n_components])       .dot(Vt[0:n_components,
 0:Vt.shape[1]])
----> 7 SSE = np.sum((A - A_hat_reconstructed)**2)
      8 comp_ratio = (A.shape[1]*n_components + n_components + A.sha
pe[0]*n_components)/(A.shape[1] * A.shape[0])
      9 

/anaconda3/lib/python3.6/site-packages/numpy/matrixlib/defmatrix.py
 in __pow__(self, other)
    320 
    321     def __pow__(self, other):
--> 322         return matrix_power(self, other)
    323 
    324     def __ipow__(self, other):

/anaconda3/lib/python3.6/site-packages/numpy/matrixlib/defmatrix.py
 in matrix_power(M, n)
    137     M = asanyarray(M)
    138     if M.ndim != 2 or M.shape[0] != M.shape[1]:
--> 139         raise ValueError("input must be a square array")
    140     if not issubdtype(type(n), N.integer):
    141         raise TypeError("exponent must be an integer")

ValueError: input must be a square array
```

In [15]:

```
"""
#if Eigenvalues have not been sorted somehow - not relevant to np.linalg.svd
n_components = 50

U, s, Vt = np.linalg.svd(A, full_matrices=False)
S = np.diag(s)

q = np.argsort(np.diag(S))[::-1] #get positions of sorted (ascending) array and
 reverse (to get descenting)
sig = q[:n_components] #positions of dominant slices

A_hat_reconstructed = U[np.ix_(np.arange(U.shape[0]), sig)]\
    .dot(S[np.ix_(sig, sig)]\
    .dot(Vt[np.ix_(sig, np.arange(Vt.shape[1]))]) )
SSE = np.sum((A - A_hat_reconstructed)**2)
comp_ratio = (A.shape[1]*n_components + n_components + A.shape[0]*n_components)/
(A.shape[1] * A.shape[0])

print('If we choose {} dominant singular value/s;\n SSE = {} and\n compresion ra
tio = {}\n\n'\
      .format(n_components, SSE, np.round(comp_ratio,10)))
"""
```

Out[15]:

```
"\n#if Eigenvalues have not been sorted somehow - not relevant to n
p.linalg.svd\nn_components = 50\n\nU, s, Vt = np.linalg.svd(A, full_
matrices=False)\nS = np.diag(s)\n\nq = np.argsort(np.diag(S))[::-1]
#get positions of sorted (ascending) array and reverse (to get desce
nting)\nsig = q[:n_components] #positions of dominant slices\n\nA_ha
t_reconstructed = U[np.ix_(np.arange(U.shape[0]), sig)]   .dot(S[np.
ix_(sig, sig)]   .dot(Vt[np.ix_(sig, np.arange(Vt.shape[1]))]) )\nSS
E = np.sum((A - A_hat_reconstructed)**2) \ncomp_ratio = (A.shape[1]*
n_components + n_components + A.shape[0]*n_components)/(A.shape[1] *
A.shape[0])\n\nprint('If we choose {} dominant singular value/s;\n S
SE = {} and\n compresion ratio = {}\n\n'      .format(n_components,
SSE, np.round(comp_ratio,10)))\n"
```

In [16]:

```
pl.figure(figsize=(15,10)) #figsize=(15,10)
pl.subplot(121)
pl.imshow(A, cmap=pl.cm.gray)
pl.title('Original image')
pl.subplot(122)
pl.imshow(A_hat_reconstructed, cmap=pl.cm.gray)
pl.title('Compressed image')
pl.show()
```