Assignment - I

Name:	**Gavali Deshabhakt Nagnath**
Subject:	**Machine Learning**
Specialization :	**Computational and Data Sciences**
Department:	**Mathematical and Computational Sciences**
Course Instructor:	**Dr. Jidesh P.**
Data:	**14/04/2021**

Q1. Write codes to perform, LU, LDU, QR, and SV Decomposition.

A. LU - Decomposition:

Program:

```
import numpy as np
import copy

def inputMatrix():                # Function to take matrix input from user
    print("Enter the size of matrix: ")
    n= int(input())

    A= np.zeros((n,n),dtype=float)
    print("Now enter elements of matrix 'A':")
    for i in range(n):
        print("Enter elements for row:",i+1)
        for j in range(n):
            A[i][j]=int(input())
    return A

def printMatrix(V):               # Function to print Matrix
    for i in range(n):
        for j in range(n):
            print(f'{V[i][j]:15.08f}' ,  end="  ")
        print()
    print()

def LUDecomposition(A):     #LU-Decomposition function definition

    n = len(A)

    L= np.zeros((n,n),dtype=float)
    for i in range(len(L)):
        L[i][i]=1

    U = copy.copy(A)    # copying matrix A into U

    for i in range(0,n-1):
        for j in range(i+1,n):
            L[j][i]= (U[j][i]/U[i][i])
            U[j][:]=U[j][:]-L[j][i]*U[i][:]

    return L,U


# Default input
n = 3

A = np.array([
```

```
    [1,2,4],
    [3,8,14],
    [2,6,13]
])

# Uncomment below line to take input from user
# A = inputMatrix()

# Calling Function of matrix A
L,U = LUDecomposition(A)

# Printing Results
print("A = ")
printMatrix(A)

print("L = ")
printMatrix(L)

print("U = ")
printMatrix(U)
```

Output:

```
A =
    1.00000000    2.00000000     4.00000000
    3.00000000    8.00000000    14.00000000
    2.00000000    6.00000000    13.00000000

L =
    1.00000000    0.00000000    0.00000000
    3.00000000    1.00000000    0.00000000
    2.00000000    1.00000000    1.00000000

U =
    1.00000000    2.00000000    4.00000000
    0.00000000    2.00000000    2.00000000
    0.00000000    0.00000000    3.00000000
```

B. LDU – Decomposition

Program:

```
import numpy as np
import copy

def inputMatrix():                # Function to take matrix input from user
    print("Enter the size of matrix: ")
    n= int(input())

    A= np.zeros((n,n),dtype=float)
    print("Now enter elements of matrix 'A':")
```

```
    for i in range(n):
        print("Enter elements for row:",i+1)
        for j in range(n):
            A[i][j]=int(input())
    return A

def printMatrix(V):              # Function to print matrix
    for i in range(n):
        for j in range(n):

            print(f'{V[i][j]:15.08f}' ,  end="  ")
        print()
    print()


def LUDecomposition(A):              # LDU – Decomposition Function Definition

    n = len(A)

    L= np.zeros((n,n),dtype=float)
    D = np.zeros((n,n),dtype=float)

    for i in range(len(L)):
        L[i][i]=1

    U = copy.copy(A)    # copying matrix A into U



    for i in range(0,n-1):
        for j in range(i+1,n):
            L[j][i]= (U[j][i]/U[i][i])
            U[j][:]=U[j][:]-L[j][i]*U[i][:]

    for i in range(n):
        if(U[i][i]!=0):
            D[i][i] = copy.copy(U[i][i])
            U[i,:] = copy.copy(U[i,:]/U[i][i])

    return L,D,U

# Default Input
n = 3

A = np.array([
    [1,2,4],
    [3,8,14],
    [2,6,13]
])
```

```
# Calling LDU decomposition function
L, D, U = LUDecomposition(A)

print("A = ")
printMatrix(A)

print("L = ")
printMatrix(L)

print("D = ")
printMatrix(D)

print("U = ")
printMatrix(U)
```

Output:

```
A =
    1.00000000     2.00000000     4.00000000
    3.00000000     8.00000000    14.00000000
    2.00000000     6.00000000    13.00000000

L =
    1.00000000     0.00000000     0.00000000
    3.00000000     1.00000000     0.00000000
    2.00000000     1.00000000     1.00000000

D =
    1.00000000     0.00000000     0.00000000
    0.00000000     2.00000000     0.00000000
    0.00000000     0.00000000     3.00000000

U =
    1.00000000     2.00000000     4.00000000
    0.00000000     1.00000000     1.00000000
    0.00000000     0.00000000     1.00000000
```

C. QR – Decomposition

Program:

```
import numpy as np

def matrixInput():
    m = int(input("Enter row size :"))
    n = int(input("Enter column size :"))

    A = np.zeros((m,n), dtype=float)
```

```
    print("Enter elements of matrix: ")
    for i in range(m):
        for j in range(n):
            A[i][j] = float(input())

    return A


def Normalize(v):
    sum = 0.0
    for i in v:
        sum+=i**2
    v=v/(sum**0.5)
    return v

def QRDecomp(A):

    n = len(A[0])   # Columns/Vectors
    m = len(A)      # Rows/Components
    q = []

    q.append(Normalize(A[:,0].reshape(m,1)))

    for i in range(1,n):

        vec = A[:,i].astype('float64').reshape(m,1)
        temp = np.zeros((m,1),dtype=float)

        for j in range(i):

            multiplier = (((vec.transpose()).dot((q[j]))))/(q[j].transpose().dot(q[j]))
            temp -= (multiplier)*q[j]

        vec = vec + temp
        normalizedvec = Normalize(vec)
        q.append(normalizedvec)

    Q = np.array(q).transpose().reshape(m,n)    # typecasting python list to numpy array and taking
np.array's transpose

    # Calculating R
    R = np.zeros((n,n))
    for i in range(n):
        for j in range(n):
            if i<=j:
                R[i][j] = A[:,j].transpose().dot(Q[:,i])

    return Q,R
```

```
def printMatrix(V):
    m = len(V)
    n = len(V[0])
    for i in range(m):
        for j in range(n):
                print(f'{V[i][j]:10.05f}' ,  end="  ")
        print()
    print()

# Default Input

A = np.array(((
    (1, -1, 4),
    (1, 4, -2),
    (1, 4, 2),
    (1, -1, 0)
)))

# Uncomment following lines for custom input
# A = matrixInput()

# Calling QR-decomposition function
Q,R = QRDecomp(A)

print("A = ")
printMatrix(A)

print("Q =")
printMatrix(Q)

print("R =")
printMatrix(R)
```

Output:

```
A =
   1.00000    -1.00000     4.00000
   1.00000     4.00000    -2.00000
   1.00000     4.00000     2.00000
   1.00000    -1.00000     0.00000

Q =
   0.50000    -0.50000     0.50000
   0.50000     0.50000    -0.50000
   0.50000     0.50000     0.50000
   0.50000    -0.50000    -0.50000

R =
   2.00000     3.00000     2.00000
```

```
0.00000     5.00000    -2.00000
0.00000     0.00000     4.00000
```

D. SVD

Program:

```
# # SVD Implementation
# ## Importing libraries

# In[1]:

import numpy as np
import copy

# ## SVD function Definition

# In[2]:


def printMatrix(V):
    m = len(V)
    n = len(V[0])

    for i in range(m):
        for j in range(n):
            print(f'{V[i][j]:10.05f}' ,  end="  ")
        print()
    print()


def SVD(A):
    m = len(A)
    n = len(A[0])

    At = A.transpose()

    AtA = np.matmul(At,A)

    AAt = np.matmul(A,At)

    # Finding Eigen Values and Vectors of AAt and AtA
    eigValuesAAt, eigVectorsAAt = np.linalg.eig(AAt)

    eigValuesAtA, eigVectorsAtA = np.linalg.eig(AtA)

    # Forming U, D and VT
    U = eigVectorsAAt
```

```
# Sorting eigen values in descending order and also changing position of corresponding eigen
vectors
    idx = eigValuesAAt.argsort()[::-1]
    eigValuesAAt = eigValuesAAt[idx]
    eigVectorsAAt = eigVectorsAAt[:,idx]
    eigVectorsAtA = eigVectorsAtA[:,idx]


    D = np.zeros((m,n))
    for i in range(m):
        for j in range(n):
            if i==j:
                D[i][j] = (eigValuesAAt[i])**(1/2)
            else:
                D[i][j] = 0

    Vt = eigVectorsAtA.transpose()

    return U,D,Vt


# In[3]:


A = np.array(((
    (1,2,3),
    (4,5,6),
    (7,8,9)
)))

# A = np.array(((
#     (1, -1, 4),
#     (1, 4, -2),
#     (1, 4, 2),
#     (1, -1, 0)
# )))

# Calling SVD-decomposition function
U,D,Vt = SVD(A)


# In[4]:

print("A = ")
printMatrix(A)
print("U = ")
printMatrix(U)
print("D = ")
printMatrix(D)
```

```
print("VT = ")
printMatrix(Vt)
```

Output:

```
A =
   1.00000     2.00000     3.00000
   4.00000     5.00000     6.00000
   7.00000     8.00000     9.00000

U =
  -0.21484    -0.88723     0.40825
  -0.52059    -0.24964    -0.81650
  -0.82634     0.38794     0.40825

D =
  16.84810     0.00000     0.00000
   0.00000     1.06837     0.00000
   0.00000     0.00000     0.00000

VT =
  -0.47967    -0.57237    -0.66506
  -0.77669    -0.07569     0.62532
   0.40825    -0.81650     0.40825
```

Q2.1 PCA of Yale Face Database

Importing Libraries

In [1]:
```
import numpy as np
from matplotlib.image import imread
import matplotlib.pyplot as plt
import scipy.io
import copy

plt.rcParams['figure.figsize'] = [8,4]
```

Importing Yale Faces Database from .mat file

scipy.io.loadmat() function imports .mat file as dictonary

In [2]:
```
data = scipy.io.loadmat('./Yale_64x64.mat')
print(type(data))
```
```
<class 'dict'>
```

Dictonary to Numpy Array

In [3]:
```
A = np.array(data['fea']).T
```

In [4]:
```
print(A.shape)
```
```
(4096, 165)
```

Sample Image/Face from database

In [5]:
```
img = plt.imshow(A[:,1].reshape(64,64).transpose())
img.set_cmap('gray')
plt.axis('off')
plt.show()
```

Forming Covariance-Matrix

In [6]:
```
Amean = A.mean(axis=1,keepdims=True)
Am = A - Amean
```

In [7]:
```
img = plt.imshow(Am[:,1].reshape(64,64).transpose())
img.set_cmap('gray')
plt.axis('off')
plt.show()
```

Calculating SVD

In [8]:
```
U,D,Vt = np.linalg.svd(Am)        # Complete SVD i.e. calculation corresponding to ze

# U,D,Vt = np.linalg.svd(Am, full_matrices=False)    # Economy SVD i.e. Calculations c

D = np.diag(D)
```

In [9]:
```
print(U.shape,D.shape,Vt.shape)
```
```
(4096, 4096) (165, 165) (165, 165)
```

Finding number of eigen values with least significance

In [10]:
```
i = 0
n = len(D)
while(i<n):
    if abs(D[i][i]) <10:
        break
    i += 1
eig_vals_with_least_significance = n - i
```

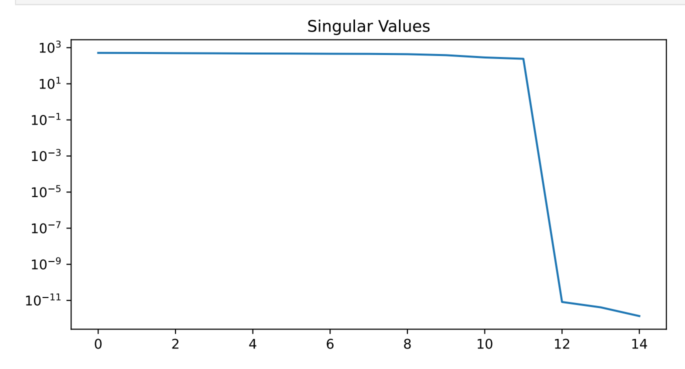
In [11]:
```
print(eig_vals_with_least_significance)
```
```
3
```

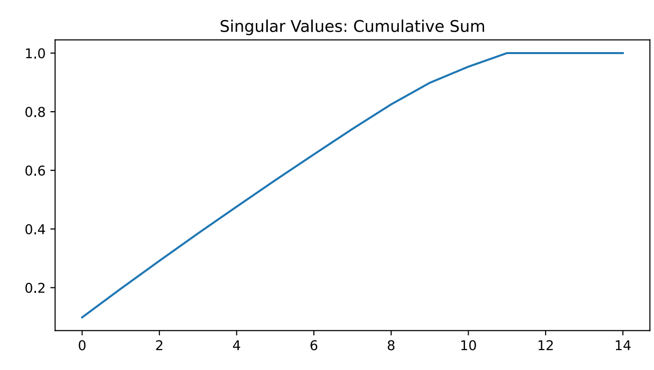
Visualizing singular values by plotting graph

1. Singular values vs Count

2. (Cumulative sum/Total sum) vs Count

In [12]:
```
d = D[150:,150:]

plt.figure(1)
plt.semilogy(np.diag(d))
plt.title('Singular Values')
plt.show()

plt.figure(2)
plt.plot(np.cumsum(np.diag(d))/np.sum(np.diag(d)))
plt.title('Singular Values: Cumulative Sum')
plt.show()
```

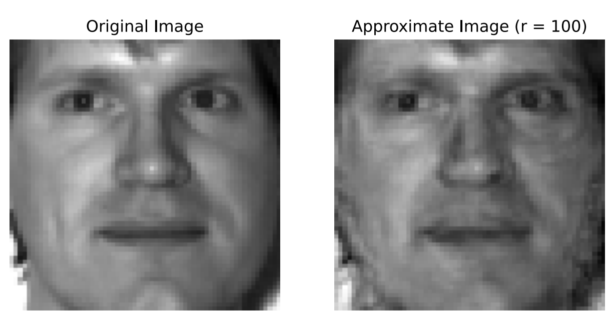


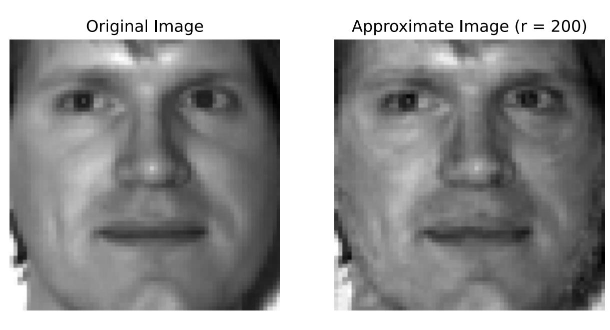


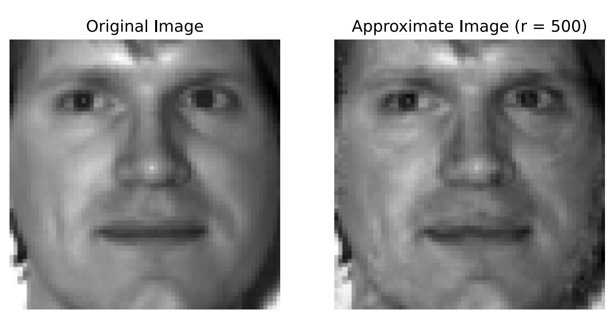
In Sample Projection and Prediction

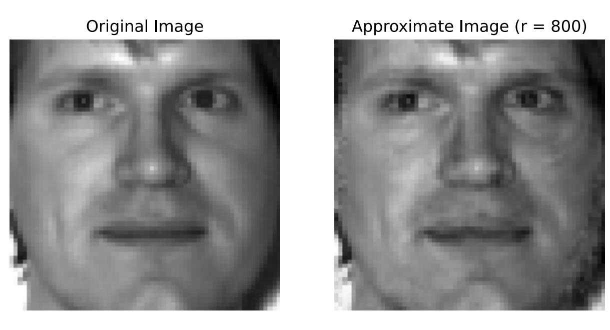
In [13]:
```
sample_size = 150

def InSampleProjectionAndReconstruction(image_number):

    j = 0

    for r in (50, 100, 200, 500, 800, 2000, 4096, 4096-eig_vals_with_least_significanc

        # Construct approximate image
        u = U[:,:r]

        # Projection
        A_train_model = np.matmul(u.T,A[:,:sample_size])

        # Reconstruction
        A_train_pred = np.matmul(u,A_train_model)
        Fimg = A_train_pred

        plt.figure(j+1)
        j += 1

        plot1 = plt.subplot(121)
        img =  plt.imshow(A[:,image_number].reshape(64,64).transpose())
        img.set_cmap('gray')
        plt.title(f'Original Image')
        plt.axis('off')

        plot2 = plt.subplot(122)
        img2 = plt.imshow(Fimg[:,image_number].reshape(64,64).transpose())
        img2.set_cmap('gray')
        plt.axis('off')
        plt.title(f'Approximate Image (r = {r})')
        plt.show()
```

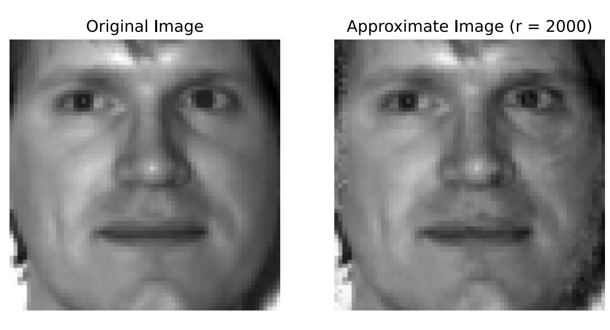
In [14]:
```
InSampleProjectionAndReconstruction(0)
```







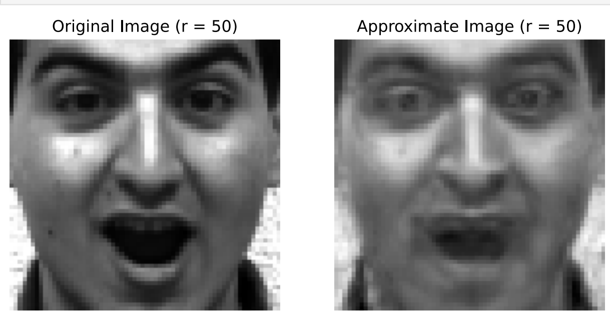


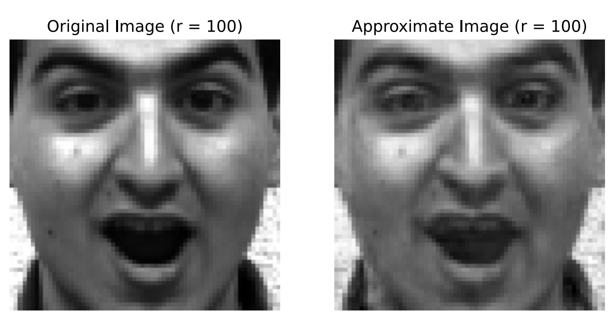


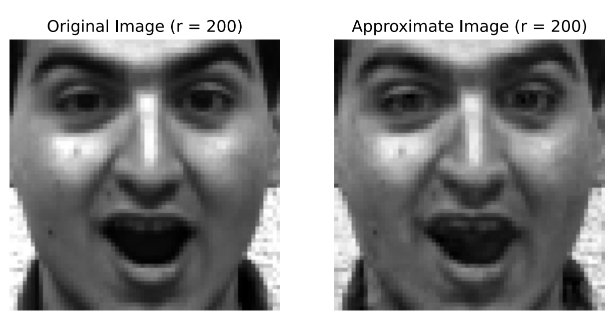
Out off Sample Projection and Prediction

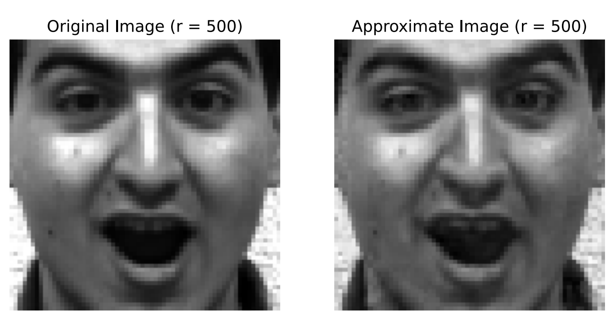
In [15]:
```
def outOffSampleProjectionAndReconstruction(image_number):
    if(image_number>=sample_size):
        j = 0
        for r in (50, 100, 200, 500, 800, 2000,4096, 4096-eig_vals_with_least_signific

            # Construct approximate image
            u = U[:,:r]

            # Projection
            A_test_model = np.matmul(u.T,A[:,image_number])

            # Reconstruction
            A_test_pred = np.matmul(u,A_test_model)
            Fimg = A_test_pred

            plt.figure(j+1)
            j += 1

            plt.subplot(121)
            img =  plt.imshow(A[:,image_number].reshape(64,64).transpose() )

            img.set_cmap('gray')
            plt.axis('off')
            plt.title(f'Original Image (r = {r})')

            plt.subplot(122)
            img2 = plt.imshow(Fimg.reshape(64,64).transpose())
            img2.set_cmap('gray')
            plt.axis('off')
            plt.title(f'Approximate Image (r = {r})')
            plt.show()

    else:
            print("Object Belongs to Sample")
```

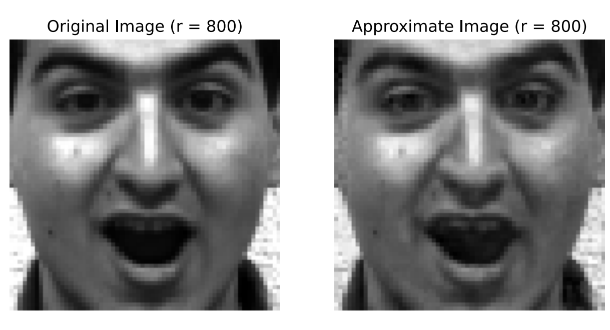
In [16]:
```
outOffSampleProjectionAndReconstruction(155)
```

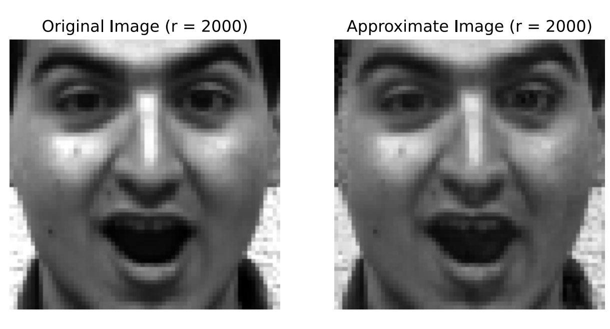


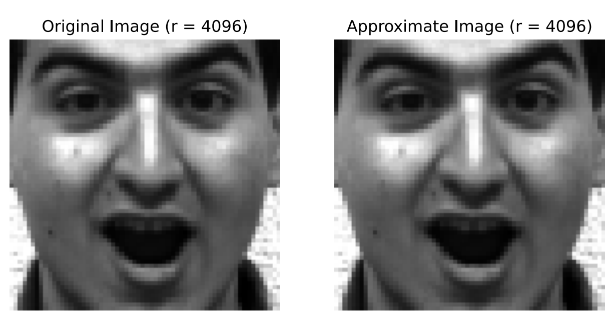


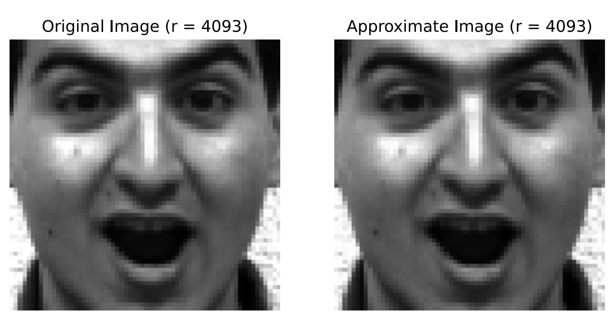












Q2.2 Dual PCA of Yale Face Database

Importing Libraries

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.io

plt.rcParams['figure.figsize'] = [10,5]
```

Importing Yale Face Dababase

In [2]:
```
data = scipy.io.loadmat('./YaleFaceDataBase/Yale_64x64.mat')
```

In Dual PCA if A has dimensions n by t then n >> t

Taking first t-number of images for training

In [3]:
```
t = 150
X = np.array(data['fea'])[:t,:].T
```

In [4]:
```
Xmean = X.mean(axis=1,keepdims=True)
Xm = X - Xmean
```

In [5]:
```
print(Xm.shape)
```

```
(4096, 150)
```

Visualizing one of the sample image

In [6]:
```
img = plt.imshow(X[:,1].reshape(64,64).transpose())
img.set_cmap('gray')
plt.axis('off')
plt.show()
```

Calculating At*A

In [7]:
```
XtX = np.matmul(Xm.T,Xm)
```

In [8]:
```
print(XtX.shape)
```

```
(150, 150)
```

Calculating Eigen values of At*A

In [9]:
```
eigValues, eigVectors = np.linalg.eigh(XtX)
```

In [10]:
```
print(eigValues)
```

```
[-9.31601916e-09 -1.81932104e-09  2.50805120e-11  6.21710919e+04
  8.59125556e+04  2.04228862e+05  2.11942004e+05  2.27778287e+05
  2.45877139e+05  2.48935034e+05  2.61768761e+05  2.81587371e+05
  2.87727134e+05  2.94493510e+05  2.99302549e+05  3.04625394e+05
  3.09637639e+05  3.23721047e+05  3.32884885e+05  3.38567033e+05
  3.57493821e+05  3.61945240e+05  3.66792327e+05  3.76439271e+05
  3.86853324e+05  3.99484472e+05  4.05011945e+05  4.18832856e+05
  4.28724230e+05  4.30747311e+05  4.43266343e+05  4.55533425e+05
  4.59853149e+05  4.70594424e+05  4.74707029e+05  4.89340006e+05
  5.16791173e+05  5.26939777e+05  5.30587323e+05  5.43556548e+05
  5.71222769e+05  5.77657089e+05  5.81025461e+05  5.94021592e+05
  6.01079311e+05  6.08513227e+05  6.16588876e+05  6.55173458e+05
  6.75456752e+05  6.87058278e+05  6.95057133e+05  7.09786552e+05
  7.23948716e+05  7.37620959e+05  7.63339470e+05  7.72923728e+05
  7.95124296e+05  8.00871360e+05  8.38725093e+05  8.66622898e+05
  8.83355570e+05  8.98292380e+05  9.06103949e+05  9.25543606e+05
  9.27096698e+05  9.54275562e+05  9.82856634e+05  1.00377235e+06
  1.01068913e+06  1.03363897e+06  1.05471367e+06  1.06496997e+06
  1.09152230e+06  1.13485347e+06  1.15647391e+06  1.17119843e+06
  1.21833494e+06  1.24221582e+06  1.27482419e+06  1.29179711e+06
  1.34801534e+06  1.39016603e+06  1.43486715e+06  1.46625796e+06
  1.49384225e+06  1.51837450e+06  1.56878060e+06  1.60236414e+06
  1.63786650e+06  1.70253015e+06  1.74280119e+06  1.77591272e+06
  1.85114296e+06  1.89244886e+06  1.93664368e+06  2.03311434e+06
  2.04949479e+06  2.17793057e+06  2.22494229e+06  2.27632881e+06
  2.36096126e+06  2.39260104e+06  2.47879183e+06  2.58928405e+06
  2.63171136e+06  2.81075972e+06  2.94613938e+06  2.97347169e+06
  3.05251301e+06  3.17320774e+06  3.50437004e+06  3.52343595e+06
  3.56516647e+06  3.84817695e+06  3.93203933e+06  4.16951502e+06
  4.48642508e+06  4.71018644e+06  5.02710731e+06  5.36278682e+06
  5.58171194e+06  5.67220922e+06  5.92610205e+06  6.34470343e+06
  6.68814138e+06  6.84549167e+06  7.37659104e+06  7.79122096e+06
  7.99443808e+06  9.27090838e+06  1.03605883e+07  1.06054777e+07
  1.16768149e+07  1.19870162e+07  1.30195907e+07  1.42187369e+07
  1.71881454e+07  1.86918862e+07  2.00183011e+07  2.46460019e+07
  2.56679223e+07  2.91281971e+07  3.51661170e+07  4.08503725e+07
  5.72544262e+07  7.47329422e+07  8.18420061e+07  1.70424206e+08
  1.94280675e+08  3.28072532e+08]
```

Sorting eigen values in descending values and changing order of

eigen vectors correspondingly

In [11]:
```
idx = eigValues.argsort()[::-1]
eigValues = eigValues[idx]
eigVectors = eigVectors[:,idx]
```

In [12]:
```
print(eigValues)
```

```
[ 3.28072532e+08  1.94280675e+08  1.70424206e+08  8.18420061e+07
  7.47329422e+07  5.72544262e+07  4.08503725e+07  3.51661170e+07
  2.91281971e+07  2.56679223e+07  2.46460019e+07  2.00183011e+07
  1.86918862e+07  1.71881454e+07  1.42187369e+07  1.30195907e+07
  1.19870162e+07  1.16768149e+07  1.06054777e+07  1.03605883e+07
  9.27090838e+06  7.99443808e+06  7.79122096e+06  7.37659104e+06
  6.84549167e+06  6.68814138e+06  6.34470343e+06  5.92610205e+06
  5.67220922e+06  5.58171194e+06  5.36278682e+06  5.02710731e+06
  4.71018644e+06  4.48642508e+06  4.16951502e+06  3.93203933e+06
  3.84817695e+06  3.56516647e+06  3.52343595e+06  3.50437004e+06
  3.17320774e+06  3.05251301e+06  2.97347169e+06  2.94613938e+06
  2.81075972e+06  2.63171136e+06  2.58928405e+06  2.47879183e+06
  2.39260104e+06  2.36096126e+06  2.27632881e+06  2.22494229e+06
  2.17793057e+06  2.04949479e+06  2.03311434e+06  1.93664368e+06
  1.89244886e+06  1.85114296e+06  1.77591272e+06  1.74280119e+06
  1.70253015e+06  1.63786650e+06  1.60236414e+06  1.56878060e+06
  1.51837450e+06  1.49384225e+06  1.46625796e+06  1.43486715e+06
  1.39016603e+06  1.34801534e+06  1.29179711e+06  1.27482419e+06
  1.24221582e+06  1.21833494e+06  1.17119843e+06  1.15647391e+06
  1.13485347e+06  1.09152230e+06  1.06496997e+06  1.05471367e+06
  1.03363897e+06  1.01068913e+06  1.00377235e+06  9.82856634e+05
  9.54275562e+05  9.27096698e+05  9.25543606e+05  9.06103949e+05
  8.98292380e+05  8.83355570e+05  8.66622898e+05  8.38725093e+05
  8.00871360e+05  7.95124296e+05  7.72923728e+05  7.63339470e+05
  7.37620959e+05  7.23948716e+05  7.09786552e+05  6.95057133e+05
  6.87058278e+05  6.75456752e+05  6.55173458e+05  6.16588876e+05
  6.08513227e+05  6.01079311e+05  5.94021592e+05  5.81025461e+05
  5.77657089e+05  5.71222769e+05  5.43556548e+05  5.30587323e+05
  5.26939777e+05  5.16791173e+05  4.89340006e+05  4.74707029e+05
  4.70594424e+05  4.59853149e+05  4.55533425e+05  4.43266343e+05
  4.30747311e+05  4.28724230e+05  4.18832856e+05  4.05011945e+05
  3.99484472e+05  3.86853324e+05  3.76439271e+05  3.66792327e+05
  3.61945240e+05  3.57493821e+05  3.38567033e+05  3.32884885e+05
  3.23721047e+05  3.09637639e+05  3.04625394e+05  2.99302549e+05
  2.94493510e+05  2.87727134e+05  2.81587371e+05  2.61768761e+05
  2.48935034e+05  2.45877139e+05  2.27778287e+05  2.11942004e+05
  2.04228862e+05  8.59125556e+04  6.21710919e+04  2.50805120e-11
 -1.81932104e-09 -9.31601916e-09]
```

In [13]:
```
eigVals = eigValues.copy()
```

Creating Singular value matrix

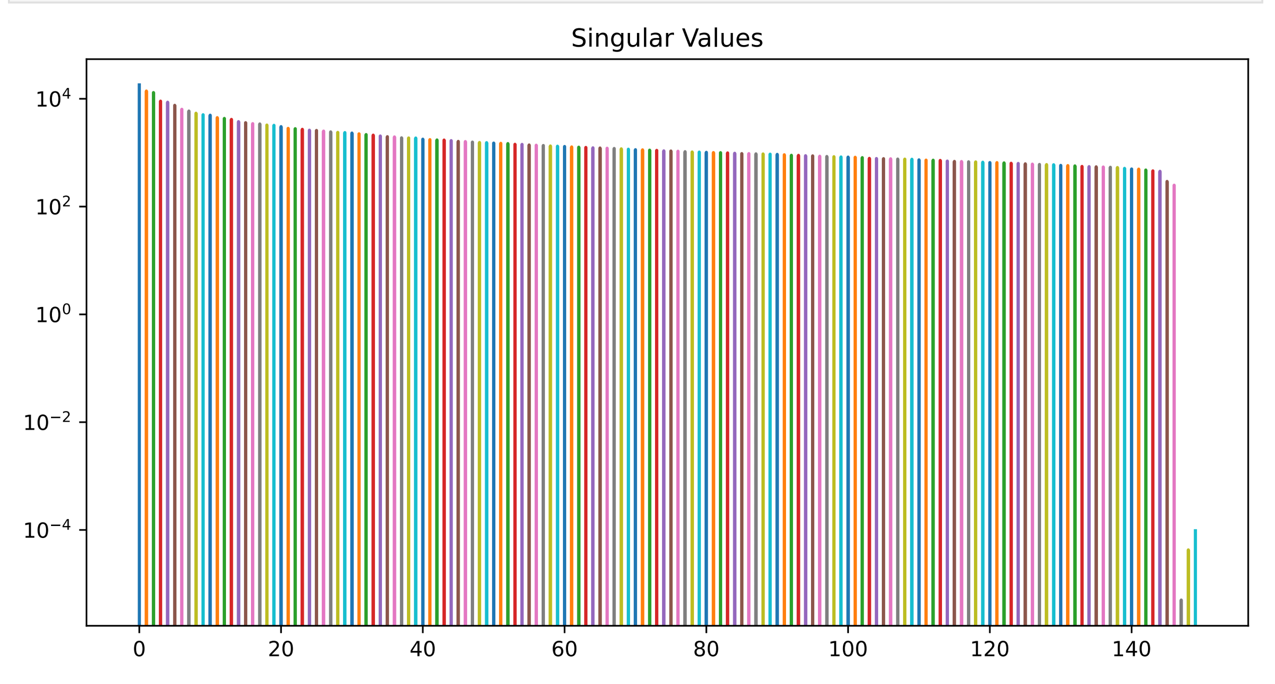
In [14]:
```
D = abs(eigVals)**0.5
D = np.diag(D)
```

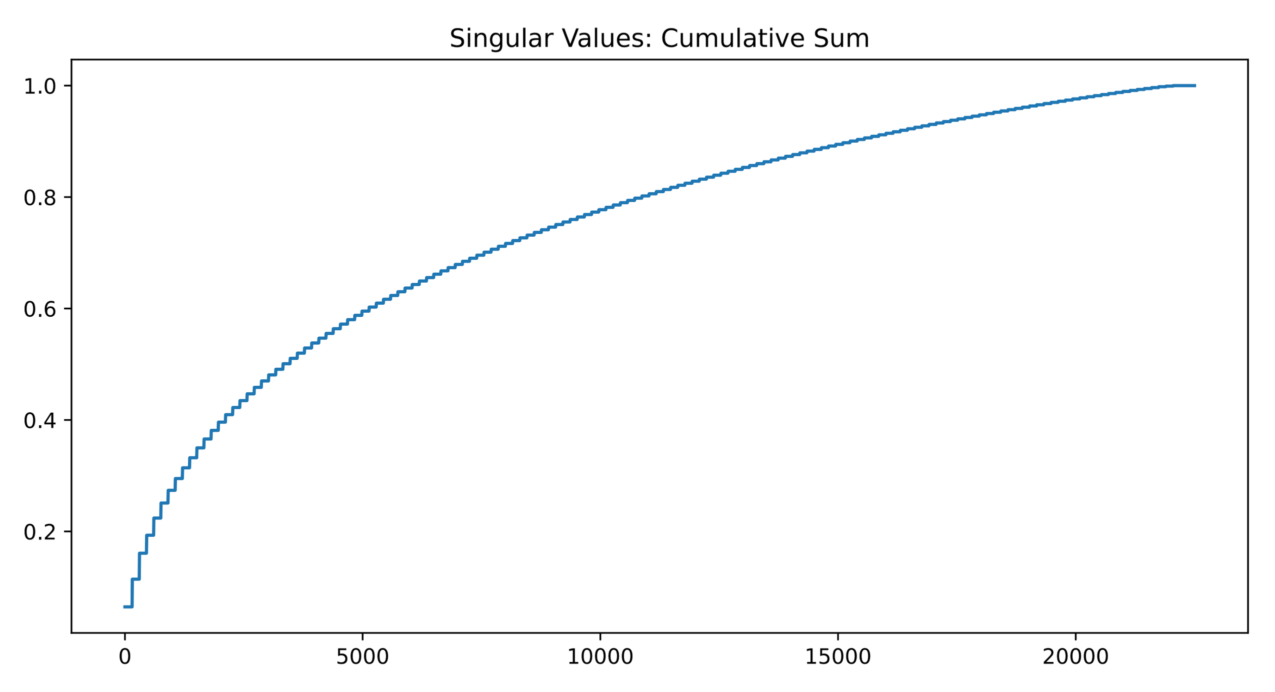
In [15]:
```
print(D)
```

```
[[1.81127726e+04 0.00000000e+00 0.00000000e+00 ... 0.00000000e+00
  0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 1.39384603e+04 0.00000000e+00 ... 0.00000000e+00
  0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 1.30546622e+04 ... 0.00000000e+00
  0.00000000e+00 0.00000000e+00]
 ...
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 ... 5.00804473e-06
  0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 ... 0.00000000e+00
  4.26534998e-05 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 ... 0.00000000e+00
  0.00000000e+00 9.65195274e-05]]
```

Visualizing Singular values matrix pattern

In [16]:
```
plt.figure(1)
plt.semilogy(D)
plt.title('Singular Values')
plt.show()
plt.figure(2)
plt.plot(np.cumsum(D)/np.sum(D))
plt.title('Singular Values: Cumulative Sum')
plt.show()
```





Formint V.transpose() Matrix

In [17]:
```
Vt = eigVectors.copy().T
```

Reconstruction of Training data

xcap = X**V**Vt

In [18]:
```
Y = D @ Vt
```

In [19]:
```
Xcap = X @ Vt.T @ np.linalg.inv(D) @ Y
```

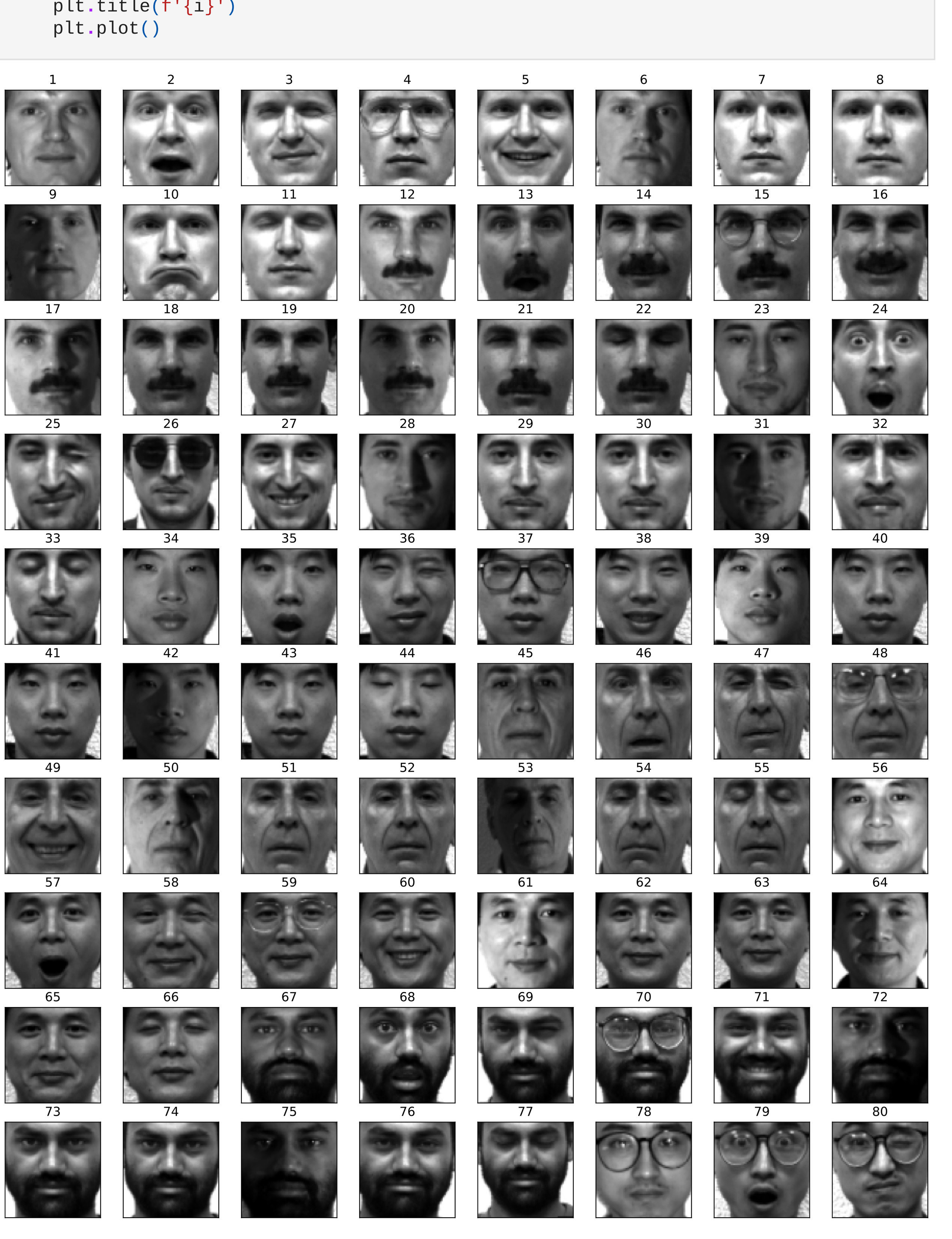
In [20]:
```
print(Xcap.shape)
```

```
(4096, 150)
```

Visualizing Reconstructed Data

In [21]:
```
img = plt.imshow(Xcap[:,5].reshape(64,64).T)
img.set_cmap('gray')
plt.axis('off')
plt.show()
```

In [22]:
```
plt.figure(figsize=(16,20))
for i in range(1,81):
    plt.subplot(10,8,i,xticks=[],yticks=[])
    img = plt.imshow(Xcap[:,i-1].reshape(64,64).T,cmap='gray')
    plt.title(f'{i}')
    plt.plot()
```



Reconstruction of Test Data

ycap = X**V**(Σ^−2)**Vt**Xtx

Calculating D inverse

In [23]:
```
invD_sq = np.linalg.inv(np.matmul(D,D))
```

Getting test data

here t is sample_size intially defined

so we're taking 165-t images as test data

In [24]:
```
X_test = np.array(data['fea'])[t:,:].T
```

Projecting test data

In [25]:
```
y = np.linalg.inv(D) @ Vt.T @ X.T @ X_test
```

Reconstructing test data

In [26]:
```
x_reconstruct = X @ Vt.T @ np.linalg.inv(D) @ y
```

Visualizing test data

In [27]:
```
img = plt.imshow(x_reconstruct[:,6].reshape(64,64).T)
img.set_cmap('gray')
plt.axis('off')
plt.show()
```

In Dual PCA, in most cases reconstruction of test data

i.e. out of sample reconstruction is not possible

Q3.1 Linear Least Square Fitting

Importing Libraries

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from prettytable import PrettyTable as ptbl
```

Importing Database

In [2]:
```
data = pd.read_csv('Salary_Data.csv')
```

In [3]:
```
data.describe()
```

Out[3]:

	YearsExperience	Salary
count	30.000000	30.000000
mean	5.313333	76003.000000
std	2.837888	27414.429785
min	1.100000	37731.000000
25%	3.200000	56720.750000
50%	4.700000	65237.000000
75%	7.700000	100544.750000
max	10.500000	122391.000000

Extracting Dependent and independent data from database int X and y variables

In [4]:
```
X = data.iloc[:,0].values
y = data.iloc[:,-1].values
```

Function for Linear Least Square Fitting

y = mx + b

In [5]:
```
def linearfitting(x,y):
    n = len(x)
    x_sq_sum = sum(x**2)
    x_sum = sum(x)
    yx_sum = sum(x*y)
    y_sum = sum(y)

    A = np.array([
        [x_sq_sum,x_sum],
        [x_sum,n]
        ])

    b = np.array([
        [yx_sum],
        [y_sum]
        ])

    invA = np.linalg.inv(A)
    M = np.matmul(invA,b)

    return M
```

Calling Linear Least Square fitting function on given database

In [6]:
```
M = linearfitting(X,y)
m = M[0][0]
b = M[1][0]
```

Visualizing Calculated Coefficient and constant

In [7]:
```
print("m = ",m,"\tb = ",b)
```

```
m =  9449.962321455096   b =  25792.200198668637
```

Calculating Approximate Values

In [8]:
```
y_pred = m*X + b
```

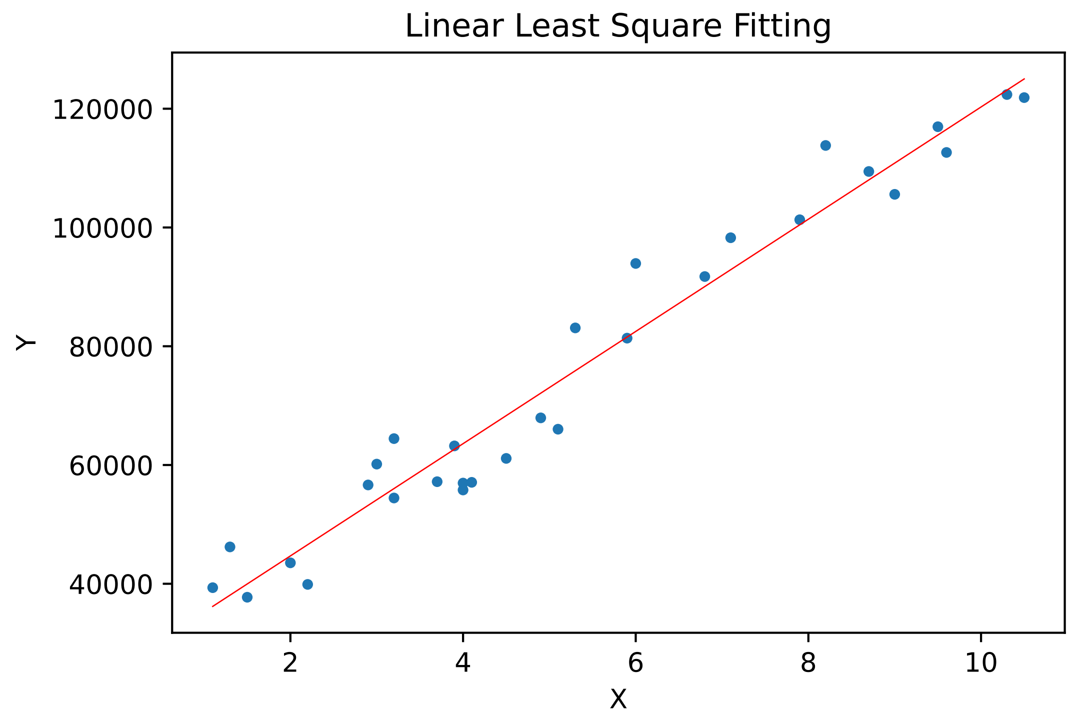
Table of actual values and predicted values

In [9]:
```
table = ptbl(['X','y','y-predicted'])
for i in range(len(X)):
    table.add_row([X[i],y[i],y_pred[i]])
print(table)
```

```
+------+----------+--------------------+
|  X   |    y     |    y-predicted     |
+------+----------+--------------------+
| 1.1  | 39343.0  | 36187.15875226924  |
| 1.3  | 46205.0  | 38077.15121656026  |
| 1.5  | 37731.0  | 39967.14368085128  |
| 2.0  | 43525.0  | 44692.12484157883  |
| 2.2  | 39891.0  | 46582.11730586985  |
| 2.9  | 56642.0  | 53197.090930888415 |
| 3.0  | 60150.0  | 54142.087163033924 |
| 3.2  | 54445.0  | 56032.07962732494  |
| 3.2  | 64445.0  | 56032.07962732494  |
| 3.7  | 57189.0  | 60757.06078805249  |
| 3.9  | 63218.0  | 62647.05325234351  |
| 4.0  | 55794.0  | 63592.04948448902  |
| 4.0  | 56957.0  | 63592.04948448902  |
| 4.1  | 57081.0  | 64537.04571663453  |
| 4.5  | 61111.0  | 68317.03064521657  |
| 4.9  | 67938.0  |  72097.0155737986  |
| 5.1  | 66029.0  | 73987.00803808963  |
| 5.3  | 83088.0  | 75877.00050238064  |
| 5.9  | 81363.0  |  81546.9778952537  |
| 6.0  | 93940.0  | 82491.97412739921  |
| 6.8  | 91738.0  | 90051.94398456329  |
| 7.1  | 98273.0  | 92886.93268099982  |
| 7.9  | 101302.0 | 100446.9025381639  |
| 8.2  | 113812.0 | 103281.89123460042 |
| 8.7  | 109431.0 | 108006.87239532797 |
| 9.0  | 105582.0 | 110841.8610917645  |
| 9.5  | 116969.0 | 115566.84225249205 |
| 9.6  | 112635.0 | 116511.83848463756 |
| 10.3 | 122391.0 | 123126.81210965612 |
| 10.5 | 121872.0 | 125016.80457394714 |
+------+----------+--------------------+
```

Visualizing Best Fit Line

In [10]:
```
plt.scatter(X,y, marker = '.')
plt.plot(X,y_pred,color = 'red',linewidth = 0.5)
plt.title('Linear Least Square Fitting')
plt.xlabel('X')
plt.ylabel('Y')
plt.show()
```



Evaluating Error in reconstruction

In [11]:
```
max_error = max(abs(y-y_pred)/y)
print(max_error)
```

```
0.17590842513666785
```

Q3.2 Quadratic Least Square Fitting

Importing Libraries

In [1]:

```python
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from prettytable import PrettyTable as ptbl
```

Importing database

In [2]:

```python
data = pd.read_csv('Quadratic_curve_fitting_dataset.csv')
```

Visualizing database

In [3]:

```python
data.head()
```

Out[3]:

	x	y
0	2	5
1	5	140
2	8	455
3	11	950
4	14	1625

Extracing Dependent and independent variables from database in X and y variables respectively

In [4]:

```python
X = data.iloc[:,0].values
y = data.iloc[:,1].values
```

Quadratic Least square fitting function

y = c1x^2 + c2x + c3

In [5]:

```python
def QuadraticFitting(x,y):

    x_four_sum = sum(x**4)
    x_three_sum = sum(x**3)
    x_sq_sum = sum(x**2)
    x_sum = sum(x)
    n = len(x)

    y_xsq_sum = sum(y*(x**2))
    yx_sum = sum(x*y)
    y_sum = sum(y)

    A = np.array([
        [x_four_sum,  x_three_sum,  x_sq_sum],
        [x_three_sum, x_sq_sum,     x_sum],
        [x_sq_sum,    x_sum,        n],
        ])

    b = np.array([
        [y_xsq_sum],
        [yx_sum],
        [y_sum]
        ])

    invA = np.linalg.inv(A)
    M = np.matmul(invA,b)

    return M
```

Calling Quadratic least square fitting function on given database

In [6]:

```python
c1, c2, c3 = QuadraticFitting(X,y)
```

Visualizing coefficients and constants

In [7]:

```python
print(f"c1 = {c1}\tc2 = {c2}\tc3 = {c3}")
```

```
c1 = [10.]      c2 = [-25.]     c3 = [15.]
```

Calculating Approximate Values

In [8]:

```python
y_pred = c1*(X**2) + c2*X + c3
```

Table of actual values and predicted values

In [9]:

```python
table = ptbl(['X','y','y-predicted'])
for i in range(len(X)):
    table.add_row([X[i],y[i],y_pred[i]])
print(table)
```

```
+-----+--------+--------------------+
|  X  |   y    |    y-predicted     |
+-----+--------+--------------------+
|  2  |   5    | 5.000000000204238  |
|  5  |  140   | 140.00000000016325 |
|  8  |  455   | 455.000000001246   |
|  11 |  950   | 950.000000000882   |
|  14 |  1625  | 1625.0000000000541 |
|  17 |  2480  | 2480.0000000000223 |
|  20 |  3515  | 3514.9999999999927 |
|  23 |  4730  | 4729.999999999965  |
|  26 |  6125  | 6124.999999999941  |
|  29 |  7700  | 7699.999999999918  |
|  32 |  9455  | 9454.999999999898  |
|  35 | 11390  | 11389.99999999988  |
|  38 | 13505  | 13504.999999999865 |
|  41 | 15800  | 15799.99999999985  |
|  44 | 18275  | 18274.99999999984  |
|  47 | 20930  | 20929.999999999833 |
|  50 | 23765  | 23764.999999999825 |
|  53 | 26780  | 26779.99999999982  |
|  56 | 29975  | 29974.999999999818 |
|  59 | 33350  | 33349.99999999982  |
|  62 | 36905  | 36904.999999999825 |
|  65 | 40640  | 40639.999999999825 |
|  68 | 44555  | 44554.99999999983  |
|  71 | 48650  | 48649.99999999985  |
|  74 | 52925  | 52924.999999999854 |
|  77 | 57380  | 57379.99999999987  |
|  80 | 62015  | 62014.99999999988  |
|  83 | 66830  | 66829.99999999991  |
|  86 | 71825  | 71824.99999999993  |
|  89 | 77000  | 76999.99999999996  |
|  92 | 82355  | 82354.99999999997  |
|  95 | 87890  |      87890.0       |
|  98 | 93605  | 93605.00000000003  |
| 101 | 99500  | 99500.00000000007  |
| 104 | 105575 | 105575.0000000001  |
| 107 | 111830 | 111830.00000000015 |
| 110 | 118265 | 118265.00000000017 |
| 113 | 124880 | 124880.00000000022 |
| 116 | 131675 | 131675.00000000026 |
| 119 | 138650 | 138650.0000000003  |
| 122 | 145805 | 145805.00000000035 |
| 125 | 153140 | 153140.0000000004  |
| 128 | 160655 | 160655.00000000047 |
| 131 | 168350 | 168350.00000000052 |
| 134 | 176225 | 176225.00000000058 |
| 137 | 184280 | 184280.00000000064 |
| 140 | 192515 | 192515.0000000007  |
| 143 | 200930 | 200930.00000000076 |
| 146 | 209525 | 209525.00000000084 |
| 149 | 218300 | 218300.00000000093 |
+-----+--------+--------------------+
```

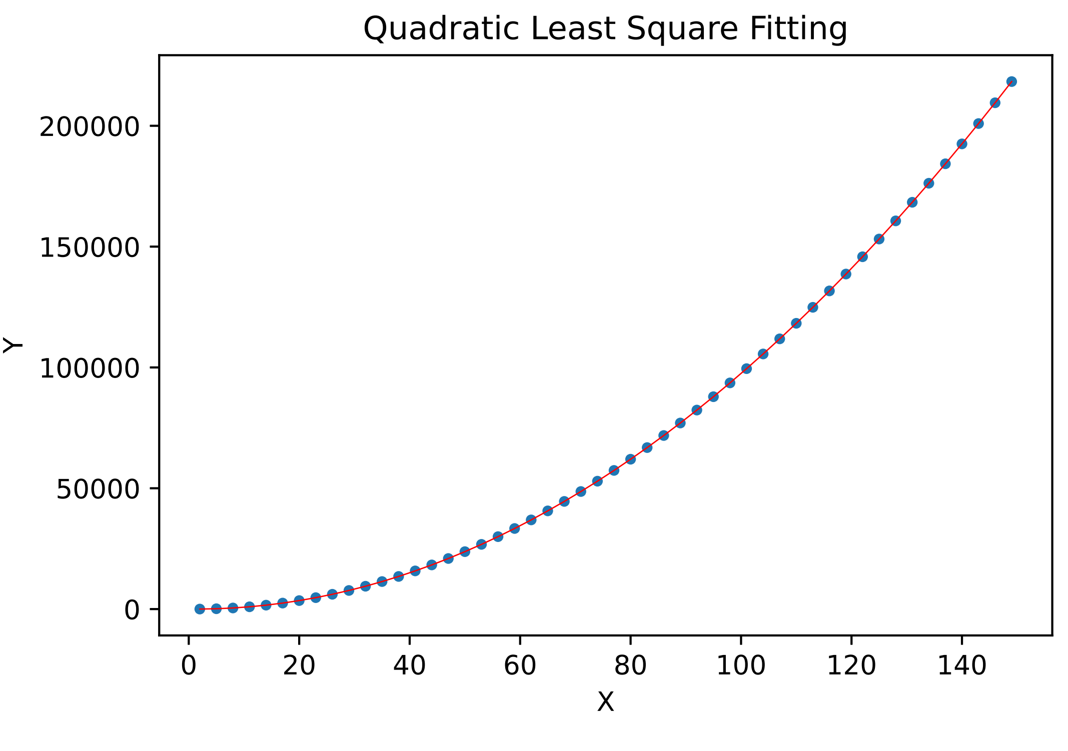
Visualizing Best Fit Curve

Note: The database used here was generated by me using Microsoft EXCEL

that's why the actual points are perfectly overlapping with approximate line

In [10]:

```python
plt.scatter(X,y, marker = '.')
plt.plot(X,y_pred,color = 'red',linewidth = 0.5)
plt.title('Quadratic Least Square Fitting')
plt.xlabel('X')
plt.ylabel('Y')
plt.show()
```



Evaluating Error in reconstruction

In [11]:

```python
max_error = max(abs(y-y_pred)/y)
print(max_error)
```

```
4.084768079337664e-11
```

Error is less because the data was generate using excel

In [12]:

```python
for i in range(5):
    print(f"y[{i}] = {y[i]}\ty_predict[{i}] = {y_pred[i]}")
```

```
y[0] = 5        y_predict[0] = 5.000000000204238
y[1] = 140      y_predict[1] = 140.00000000016325
y[2] = 455      y_predict[2] = 455.000000001246
y[3] = 950      y_predict[3] = 950.000000000882
y[4] = 1625     y_predict[4] = 1625.0000000000541
```

Q4. Denoising Using L2-Regularisation

In [1]:
```
from matplotlib.image import imread
import matplotlib.pyplot as plt
import numpy as np
import os
plt.rcParams['figure.figsize'] = [12,6]
```

Importing and Visualizing input image

In [2]:
```
Oimg = imread('dog.jpg')
print(Oimg.shape)
img = plt.imshow(Oimg)
plt.axis('off')
img.set_cmap('gray')
plt.title("Original Image")
```

```
(2000, 1500, 3)
```

Out[2]:
```
Text(0.5, 1.0, 'Original Image')
```

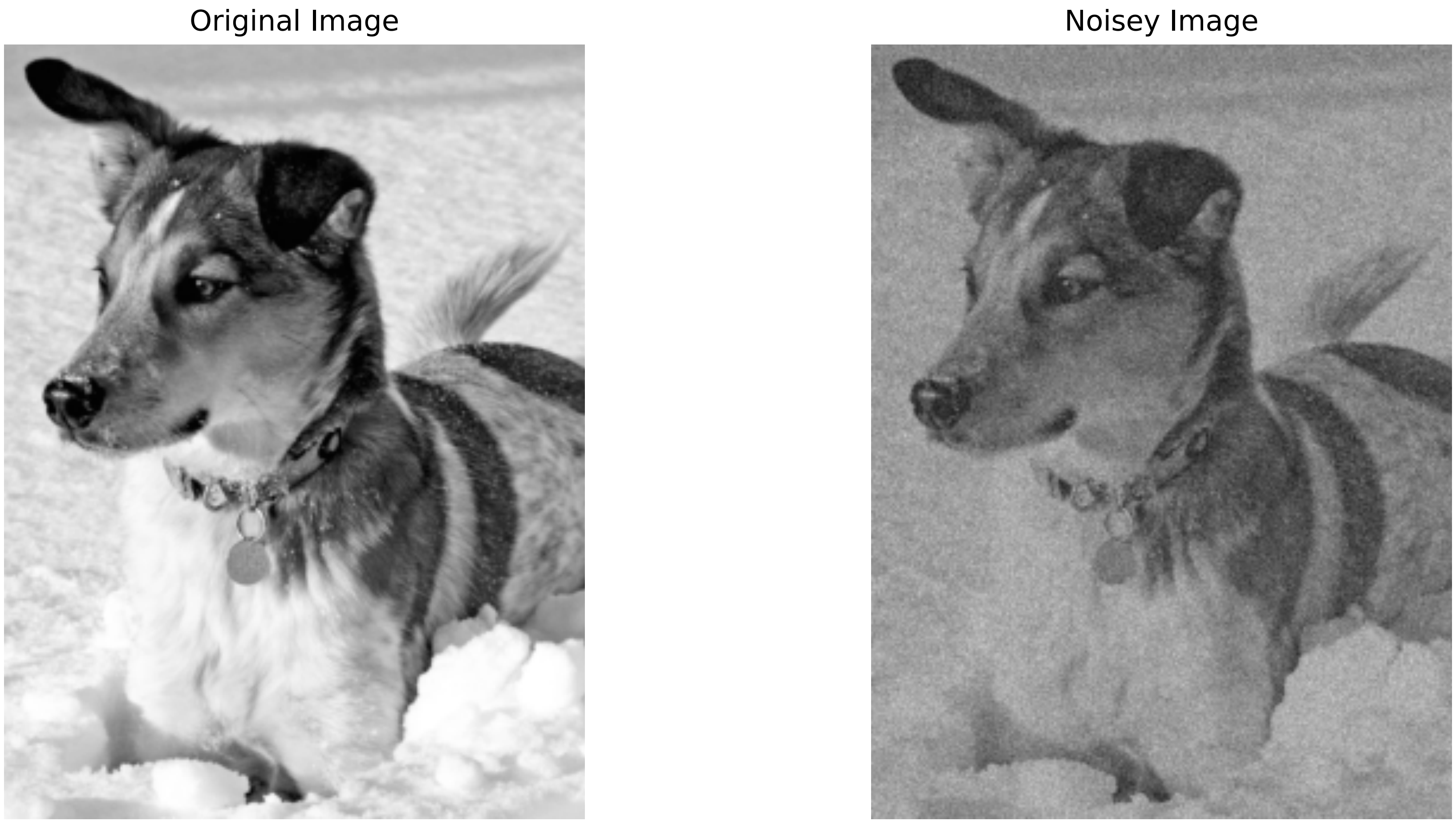
In [3]:
```
Oimg = np.mean(Oimg,-1)                 # Converting to Grayscale
```

Adding Gaussian Noise

In [4]:
```
mean = 0
sigma = 3

Noise = np.random.normal(mean, sigma, (Oimg.shape[0],Oimg.shape[1])).astype('uint8')
OimgNoisey = Oimg + Noise            # Add some noise
```

Visualizing Noise and original image

In [5]:
```
plt.figure(1)
plt.subplot(121)
img = plt.imshow(Oimg)
plt.axis('off')
img.set_cmap('gray')
plt.title("Original Image")

plt.subplot(122)
img2 = plt.imshow(OimgNoisey)
plt.axis('off')
img2.set_cmap('gray')
plt.title("Noisey Image")
plt.show()
```



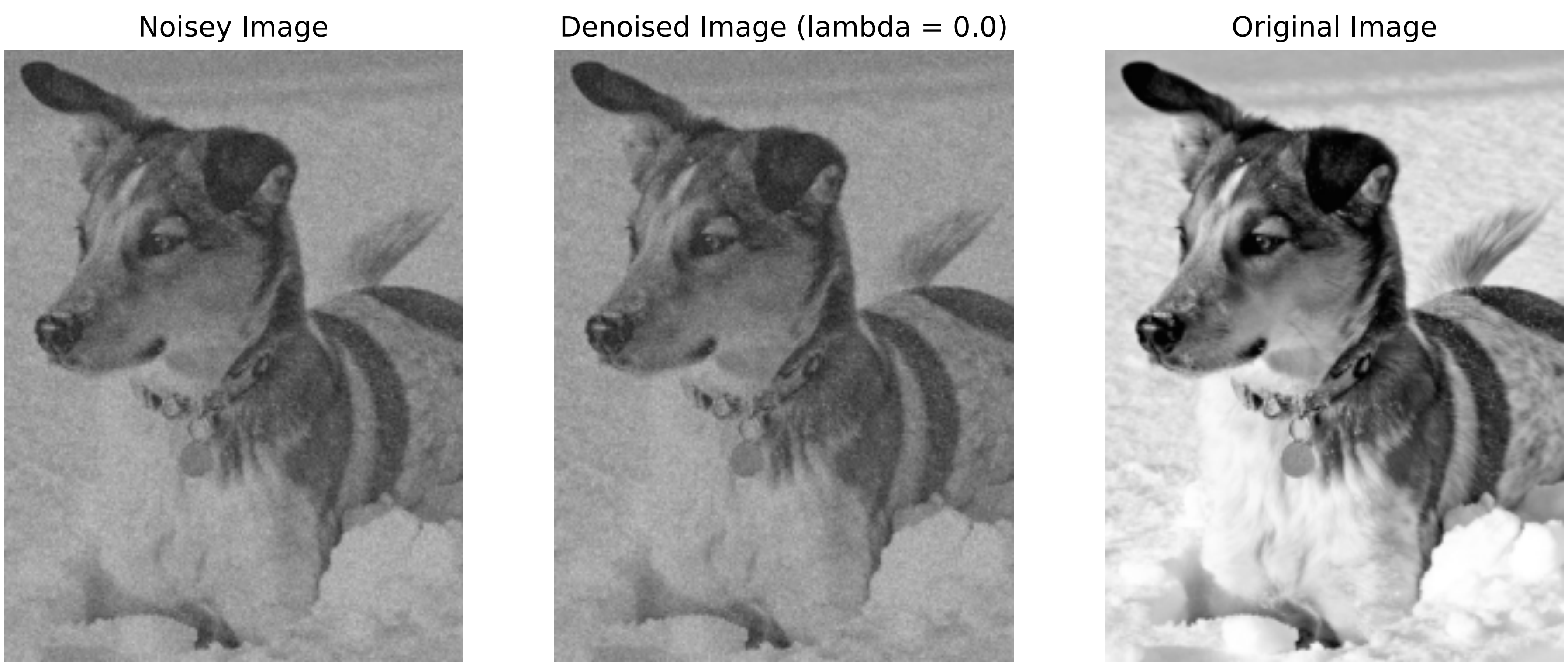
L2-regularisation Function

In [6]:

```
def L2Regularisation(NoiseyInput, ExpectedOutput,factor):

    n = len(ExpectedOutput)
    I = np.identity(n)
    A = I
    At = A.T
    AtA = np.matmul(At,A)
    M = (AtA - factor*I)

    T = np.matmul(np.linalg.inv(M),At)
    pred = np.matmul(T,NoiseyInput)

    plt.figure()
    plt.subplot(131)
    img1 = plt.imshow(NoiseyInput)
    img1.set_cmap('gray')
    plt.axis('off')
    plt.title(f'Noisey Image')

    plt.subplot(132)
    img2 = plt.imshow(pred)
    img2.set_cmap('gray')
    plt.axis('off')
    plt.title(f'Denoised Image (lambda = {factor})')

    plt.subplot(133)
    img3 = plt.imshow(ExpectedOutput)
    img3.set_cmap('gray')
    plt.axis('off')
    plt.title('Original Image')
    plt.show()
```

In [7]:

```
fact = np.arange(0,1,0.2)
for i in fact:
    L2Regularisation(OimgNoisey,Oimg,i)
```




Noisey Image
Denoised Image (lambda = 0.2)
Original Image
Noisey Image
Denoised Image (lambda = 0.4)
Original Image
Noisey Image
Denoised Image (lambda = 0.6000000000000001)
Original Image

Q5 Linear Discriminant Analysis of IRIS database

Iris dataset contains 4-features (i.e. Sepal length, Sepal width, Petal length, Petal Width)

of flowers of 3 - species (i.e. Setosa, Versicolor, Virginica)

Importing Libraries

In [1]:

```python
import math
import copy
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import LabelEncoder
```

Importing IRIS Dataset

In [2]:

```python
dataset = pd.read_csv('Iris.csv')
X = dataset.iloc[:,1:-1].values
y = dataset.iloc[:,-1].values
```

Encoding Class Labels using sklearn's labe encoder

In [3]:

```python
y_prev = copy.copy(y)

le = LabelEncoder()
le_y = le.fit(y)
y = le_y.transform(y) + 1

y_prev_vs_y_encoded = np.hstack((y_prev.reshape(len(y_prev),1), y.reshape(len(y),1)))
print(y_prev_vs_y_encoded)
```

```
[['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-setosa' 1]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-versicolor' 2]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]
 ['Iris-virginica' 3]]
```

Creating Feature and Class Label Dictonaries

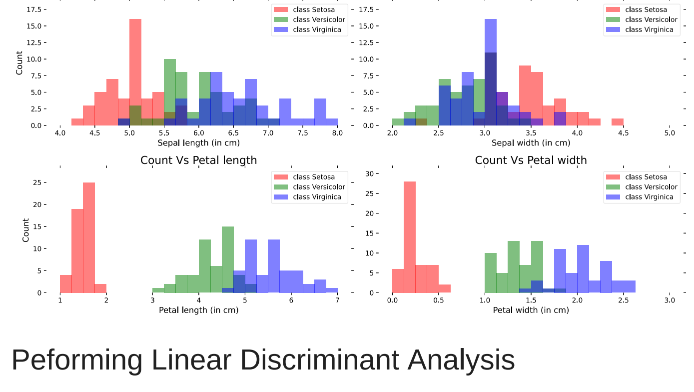
In [4]:

```python
label_dict = {1: 'Setosa', 2: 'Versicolor', 3:'Virginica'}

feature_dict = {i:label for i,label in zip(
                range(4),
                  ('Sepal length',
                  'Sepal width',
                  'Petal length',
                  'Petal width', ))}
```

Visualizing Dataset using histogram

In [5]:

```python
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(14, 7))

for ax,cnt in zip(axes.ravel(), range(4)):

    # set bin sizes
    min_b = math.floor(np.min(X[:,cnt]))
    max_b = math.ceil(np.max(X[:,cnt]))
    bins = np.linspace(min_b, max_b, 25)

    # plottling the histograms
    for lab,col in zip(range(1,4), ('red', 'green', 'blue')):
        ax.hist(X[y==lab, cnt],
                   color=col,
                   label=f'class {label_dict[lab]}',
                   bins=bins,
                   alpha=0.5,)
    ylims = ax.get_ylim()

    # plot annotation
    leg = ax.legend(loc='upper right', fancybox=True, fontsize=10)
    leg.get_frame().set_alpha(0.5)
    ax.set_ylim([0, max(ylims)+2])
    ax.set_xlabel(feature_dict[cnt]+' (in cm)', fontsize=12)
    ax.set_title(f'Count Vs {feature_dict[cnt]}', fontsize=16)

    # hide axis ticks
    ax.tick_params(axis="both", which="both", bottom="off", top="off",
            labelbottom="on", left="off", right="off", labelleft="on")

    # remove axis spines
    ax.spines["top"].set_visible(False)
    ax.spines["right"].set_visible(False)
    ax.spines["bottom"].set_visible(False)
    ax.spines["left"].set_visible(False)

axes[0][0].set_ylabel('Count', fontsize=12)
axes[1][0].set_ylabel('Count', fontsize=12)

fig.tight_layout()

plt.show()
```



Peforming Linear Discriminant Analysis

Step 1: Computing the d-dimensional mean vectors

In [6]:

```python
np.set_printoptions(precision = 4)

mean_vectors = []
for col in range(1,4):
    mean_vectors.append(np.mean(X[y==col], axis=0))
    print(f'Mean Vector class {col}: {mean_vectors[col-1]}\n')
```

```
Mean Vector class 1: [5.006 3.418 1.464 0.244]

Mean Vector class 2: [5.936 2.77  4.26  1.326]

Mean Vector class 3: [6.588 2.974 5.552 2.026]
```

Step 2: Computing the Scatter Matrices

2.1 Within-class scatter matrix S_W

In [7]:

```python
S_W = np.zeros((4,4))
for cl,mv in zip(range(1,4), mean_vectors):
    class_sc_mat = np.zeros((4,4))                  # scatter matrix for every class
    for row in X[y == cl]:
        row, mv = row.reshape(4,1), mv.reshape(4,1) # make column vectors
        class_sc_mat += (row-mv).dot((row-mv).T)
    S_W += class_sc_mat                             # sum class scatter matrices
print('Within-class Scatter Matrix:\n', S_W)
```

```
Within-class Scatter Matrix:
 [[38.9562 13.683  24.614   5.6556]
 [13.683  17.035   8.12    4.9132]
 [24.614   8.12   27.22    6.2536]
 [ 5.6556  4.9132  6.2536  6.1756]]
```

2.2 Between-class scatter matrix S_B

In [8]:

```python
overall_mean = np.mean(X, axis=0)

S_B = np.zeros((4,4))
for i,mean_vec in enumerate(mean_vectors):
    n = X[y==i+1,:].shape[0]
    mean_vec = mean_vec.reshape(4,1) # make column vector
    overall_mean = overall_mean.reshape(4,1) # make column vector
    S_B += n * (mean_vec - overall_mean).dot((mean_vec - overall_mean).T)

print('between-class Scatter Matrix:\n', S_B)
```

```
between-class Scatter Matrix:
 [[ 63.2121 -19.534  165.1647  71.3631]
 [-19.534   10.9776 -56.0552 -22.4924]
 [165.1647 -56.0552 436.6437 186.9081]
 [ 71.3631 -22.4924 186.9081  80.6041]]
```

Step 3: Solving the generalized eigenvalue problem for the matrix (S_W^-1)*(S_B)

In [9]:

```python
eig_vals, eig_vecs = np.linalg.eig(np.linalg.inv(S_W).dot(S_B))

for i in range(len(eig_vals)):
    eigvec_sc = eig_vecs[:,i].reshape(4,1)
    print('\nEigenvector {}: \n{}'.format(i+1, eigvec_sc.real))
    print('Eigenvalue {:}: {:.2e}'.format(i+1, eig_vals[i].real))
```

```
Eigenvector 1:
[[ 0.2049]
 [ 0.3871]
 [-0.5465]
 [-0.7138]]
Eigenvalue 1: 3.23e+01

Eigenvector 2:
[[-0.009 ]
 [-0.589 ]
 [ 0.2543]
 [-0.767 ]]
Eigenvalue 2: 2.78e-01

Eigenvector 3:
[[-0.8379]
 [ 0.1696]
 [ 0.1229]
 [ 0.5041]]
Eigenvalue 3: -4.13e-15

Eigenvector 4:
[[ 0.2   ]
 [-0.3949]
 [-0.4567]
 [ 0.7717]]
Eigenvalue 4: 1.20e-14
```

Checking Eigen Value and Eigen Vector Calculations

In [10]:

```python
for i in range(len(eig_vals)):
    eigvec = eig_vecs[:,i].reshape(4,1)
    np.testing.assert_array_almost_equal(np.linalg.inv(S_W).dot(S_B).dot(eigvec),
                                         eig_vals[i] * eigvec,
                                         decimal=6, err_msg='', verbose=True)
print('ok')
```

```
ok
```

Step 4: Selecting linear discriminants for the new feature subspace

4.1. Sorting the eigenvectors by decreasing eigenvalues

In [11]:

```python
# Make a list of (eigenvalue, eigenvector) tuples
eig_pairs = [(np.abs(eig_vals[i]), eig_vecs[:,i]) for i in range(len(eig_vals))]

# Sort the (eigenvalue, eigenvector) tuples from high to low
eig_pairs = sorted(eig_pairs, key=lambda k: k[0], reverse=True)

# Visually confirm that the list is correctly sorted by decreasing eigenvalues

print('Eigenvalues in decreasing order:\n')
for i in eig_pairs:
    print(i[0])
```

```
Eigenvalues in decreasing order:

32.27195779972981
0.27756686384004264
1.1953730364935478e-14
4.1311796919088535e-15
```

In [12]:

```python
print('Variance Cotained by Eigen Values (in percentage):\n')
eigv_sum = sum(eig_vals)
for i,j in enumerate(eig_pairs):
    print('eigenvalue {0:}: {1:.2%}'.format(i+1, (j[0]/eigv_sum).real))
```

```
Variance Cotained by Eigen Values (in percentage):

eigenvalue 1: 99.15%
eigenvalue 2: 0.85%
eigenvalue 3: 0.00%
eigenvalue 4: 0.00%
```

4.2. Choosing k eigenvectors with the largest eigenvalues

In [13]:

```python
W = np.hstack((eig_pairs[0][1].reshape(4,1), eig_pairs[1][1].reshape(4,1)))
print('Matrix W:\n', W.real)
```

```
Matrix W:
 [[ 0.2049 -0.009 ]
 [ 0.3871 -0.589 ]
 [-0.5465  0.2543]
 [-0.7138 -0.767 ]]
```

Step 5: Transforming the samples onto the new subspace

In [14]:

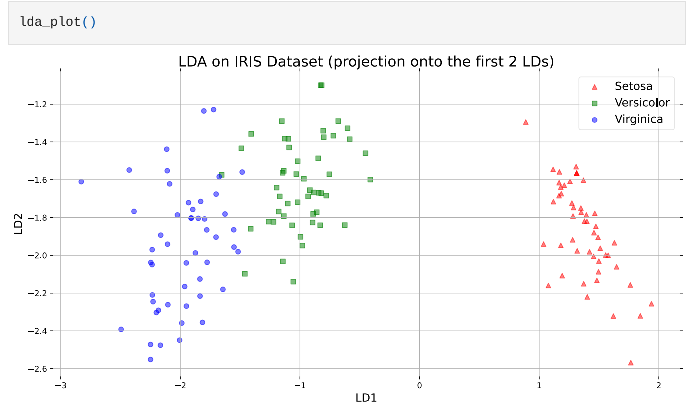
```python
X_lda = X.dot(W)
```

In [15]:

```python
def lda_plot():
    plt.rcParams['figure.figsize'] = [14,7]
    ax = plt.subplot(111)
    for label,marker,color in zip(
        range(1,4),('^', 's', 'o'),('red', 'green', 'blue')):

        plt.scatter(x=X_lda[:,0].real[y == label],
                y=X_lda[:,1].real[y == label],
                marker=marker,
                color=color,
                alpha=0.5,
                label=label_dict[label]
                )

    plt.xlabel('LD1', fontsize=14)
    plt.ylabel('LD2', fontsize=14)

    leg = plt.legend(loc='upper right', fancybox=True, fontsize=14)
    leg.get_frame().set_alpha(0.5)
    plt.title('LDA on IRIS Dataset (projection onto the first 2 LDs)', fontsize=18)

    # hide axis ticks
    plt.tick_params(axis="both", which="both", bottom="off", top="off", labelbottom="

    # remove axis spines
    ax.spines["top"].set_visible(False)
    ax.spines["right"].set_visible(False)
    ax.spines["bottom"].set_visible(False)
    ax.spines["left"].set_visible(False)

    plt.grid()
    plt.tight_layout
    plt.show()
```

In [16]:

```python
lda_plot()
```



We get perfect separation/Classification of data of all 3-classes.