1) Basic Terms :

- **Scalars** : A scalar is just a single number. Scalars are written in italics and in lower case. “Let s ∈ R be the slope of the line,” while defining a real-valued scalar, or “Let n ∈ N be the number of units,” while defining a natural number scalar.
- **Vectors** : A vector is an array of numbers arranged in specific order. We can identify each individual number by its index in that ordering. A vector could be a row vector or a column vector.Typically we give vectors lowercase names in bold typeface, such as **x**. By default Vectors are Column vectors unless it's explicitly mentioned.
- **Matrices** : A matrix is a 2-D array of numbers, so each element is identifiedby two indices instead of just one. We usually give matrices uppercasevariable names with bold typeface, such as **A**. If a real-valued matrix A has a height of m and a width of n, then we say that **A** ∈ Rm×n. A(1,1)is the upper left entry of A and A(m,n)is the bottom right entry.
- **Tensors** : An array with more than two axes is called a Tensor.In the general case, an array of numbers arranged on a regular grid with a variable number of axes is known as a tensor. We denote a tensor named “A”with this typeface: **A**. We identify the element of **A** at coordinates (i, j, k) by writing $A_{(i,j,k)}$.

In [1]:
```
import numpy as np
# Creating Vectors
row_vector = np.array([1, 2, 3, 4, 5])

column_vector = np.array([[1],
                          [2],
                          [3],
                          [4],
                          [5]])
print("Row Vector : ",row_vector,"Shape = ",row_vector.shape)
print("Column Vector: \n",column_vector,"Shape = ",column_vector.shape)
```

```
Row Vector :  [1 2 3 4 5] Shape =  (5,)
Column Vector:
 [[1]
 [2]
 [3]
 [4]
 [5]] Shape =  (5, 1)
```

In [6]:
```
#Create a Matrix
matrix = np.array([[1, 2],
                   [3, 4],
                   [5, 6]])
print("Matrix = \n",matrix,"Shape = ",matrix.shape)
```

```
Matrix =
 [[1 2]
 [3 4]
 [5 6]] Shape =  (3, 2)
```

In [15]:
```
#Create a tensor
ones_tensor = np.ones((2, 2, 2))
tensor = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
print("Tensor = \n",tensor,"Shape = ",tensor.shape)
```

```
Tensor =
 [[[ 1  2  3]
  [ 4  5  6]]

 [[ 7  8  9]
  [10 11 12]]] Shape =  (2, 2, 3)
```

- **Transpose** : The transpose of amatrix is the mirror image of the matrix across a diagonal line, called the main diagonal, running down and to the right, starting from its upper left corner. We denote the transpose of a matrix A as A^T And is defined as follows :

$$(A^T)_{i,j} = A_{j,i}$$

Vectors can be thought of as matrices that contain only one column. Thetranspose of a vector is therefore a matrix with only one row

In [150]:
```
matrix = np.array([[1, 2],
                   [3, 4],
                   [5, 6]])
#Transpose
print("Our Matrix : \n",matrix)
print("It's Transpose : \n",matrix.T)
```

```
Our Matrix :
 [[1 2]
 [3 4]
 [5 6]]
It's Transpose :
 [[1 3 5]
 [2 4 6]]
```

In [151]:
```
type(matrix)
```

Out[151]: numpy.ndarray

In [153]:
```
matrix_obj = np.matrix([[1, 2],
          [3, 4],
           [5, 6]]).T
#np.matrix cannot be used on tensors.
```

In [148]:
```
type(matrix_obj)
```

Out[148]: numpy.matrix

- **Addition Of Matrices** : We can add matrices to each other, as long as they have the same shape, just by adding their corresponding elements: **C** = **A** + **B** where $C_{i,j} = A_{i,j} + B_{i,j}$
- We can also add a scalar to a matrix or multiply a matrix by a scalar, just by performing that operation on each element of a matrix: **D** = a·**B**+c where: $D_{i,j} = a.B_{i,j} + c$
- We can also do the addition of a matrix and a vector, yielding another matrix : **C**=**A**+b,where $C_{i,j} = A_{i,j} + b_j$. Here we add vector b to each row of matrix. This is also called as broadcasting

In [45]:
```
A = np.array([[1, 2],
              [3, 4],
              [5, 6]])

B = np.array([[7, 8],
              [9, 10],
              [11, 12]])


b = np.array([[1],
              [2],
              [3]])
a = 2
c = 1
print("Sum C = \n",A+B)
print("*"*10)
print("D = \n",a*B+c)
print("*"*10)
print("Matrix Sum with vector : C = \n",A+b)
#Note: If shape of A and b doesn't match, it throws error.
```

```
Sum C =
 [[ 8 10]
 [12 14]
 [16 18]]
**********
D =
 [[15 17]
 [19 21]
 [23 25]]
**********
Matrix Sum with vector : C =
 [[2 3]
 [5 6]
 [8 9]]
```

Multiplying Matrices and Vectors

- The matrix product of matrices **A** and **B** is a third matrix **C**. Inorder for this product to be defined,**A** must have the same number of columns as **B** has rows. If **A** is of shape m × n and **B** is of shape n × p, then **C** is of shape m × p. We can write the matrix product just by placing two or more matrices together:

$$C = AB$$

The product operation is defined by :

$$C_{i,j} = \sum_k A_{i,k} B_{k,j}$$

- This is also called as the Dot Product
- **Properties Of Dot Product** :

1) Matrix Product is Distributive:

$$A(B + C) = AB + AC$$

2) Associative:

$$A(BC) = (AB)C$$

3) Not Commutative :

$$AB \neq BA$$

4) The dot product between two vectors is commutative:

$$x^T y = y^T x$$

5) The transpose of a matrix product has the form :

$$(AB)^T = B^T A^T$$

In [158]:
```
# Loops to Multiply 2 Matrices
A = np.array([[1,2,3],
              [4 ,5,6],
              [7 ,8,9]])

B = np.array([[10,11,12,13],
              [14,15,16,17],
              [18,19,20,21]])

#Multiplying 3x3 with 3x4 we get a 3x4 matrix. Initialize matrix C to zeros
C = np.zeros((3,4))


for i in range(A.shape[0]):
    for j in range(B.shape[1]):
        for k in range(B.shape[0]):
            C[i][j] += A[i][k] * B[k][j]

print("Product C(With Loops) = \n",C)

# Product of 2 matrices without loops
print("Product C(Without Loops) = \n",np.dot(A,B))
```

```
Product C(With Loops) =
 [[ 92.  98. 104. 110.]
 [218. 233. 248. 263.]
 [344. 368. 392. 416.]]
Product C(Without Loops) =
 [[ 92  98 104 110]
 [218 233 248 263]
 [344 368 392 416]]
```

- **Identity Matrix** : An identity matrix is a matrix that does not change any vector when wemultiply that vector by that matrix. All the entries along the maindiagonal are 1, while all the other entries are zero.
- **Inverse of a Matrix** : Thematrix inverseofAis denoted as A^{-1}, and it is defined as the matrix such that $A^{-1}A = I_n$.
- This can be used to solve a system of linear equations of the type Ax = b where we want to solve for x.

$$Ax = b$$

$$A^{-1}Ax = A^{-1}b$$

$$I_n x = A^{-1}b$$

Where I_n is an identity matrix of order n

$$x = A^{-1}b$$

In [79]:
```
from numpy import linalg
A = np.array([[1,8,3],
              [4,4,6],
              [9,8,9]])

b = np.array([[-10],
              [8],
              [9]])

print("Inverse of A = \n",np.linalg.inv(A))
print("Solving for x ")
x = np.linalg.solve(A,b)
print("x = \n",x)
```

```
Inverse of A =
 [[-0.1        -0.4         0.3       ]
 [ 0.15       -0.15        0.05      ]
 [-0.03333333  0.53333333 -0.23333333]]
Solving for x
x =
 [[ 0.5 ]
 [-2.25]
 [ 2.5 ]]
```

2) Linear Dependence and Span

- A set of vectors is linearly independent if no vector in a set of vectors is a linear combination of the other vectors.
- The span of a set of vectors is the set of all the points obtainable by linear combination of the original vectors.

3) Norms

- Measure the size of a vector. the L^p normis given by:

$$\|x\|_p = \left(\sum |x_i|^p\right)^{\frac{1}{p}}$$

for p ∈ R, p ≥ 1

- On an intuitive level, the norm of a vectorx measures the distance from the origin to the point x.
- The L^2 norm, with p = 2, is known as the **Euclidean norm**, which is simplythe Euclidean distance from the origin to the point identified by x.
- The L^2 norm is used so frequently in machine learning that it is often denoted simply as ||x||,with the subscript 2 omitted. It is also common to measure the size of a vectorusing the squared L^2norm, which can be calculated simply as $x^T x$. The squared L^2 norm is more convenient to work with mathematically and computationally than the L^2 norm itself.
- Each derivative of the squared L^2 norm with respect to each element of x depends only on the corresponding element of x, while all the derivatives of the L^2 norm depend on the entire vector.
- In many contexts, the squared L^2 norm may be undesirable because it increases very slowly near the origin. In these cases, we turn to a function that grows at the same rate in all locations, but that retains mathematical simplicity: the L^1 norm :

$$\|x\|_1 = \sum_i |x_i|$$

- The L^1 norm is commonly used in machine learning when the difference between zero and nonzero elements is very important. Every time an element of x movesaway from 0 by ∈, the L^1 norm increases by ∈ . The L_1 norm is often used as a substitute for the number of nonzero entries.
- L^1 matrix norm of a matrix is equal to the maximum of L^1 norm of a column of the matrix.
- **Frobenius norm** : Used to measure the size of a matrix. Also called as Euclidean norm(also used for vector L^2. norm.

$$\|A\|_F = \sqrt{\sum_{i,j} A_{i,j}^2}$$

- It is also equal to the square root of the matrix trace of $AA^{(H)}$, where A^(H) is the conjugate transpose, i.e.,

$$\|A\|_F = \sqrt{Tr(AA^H)}$$

- Where A^H is the conjugate transpose.
- Trace of a square matrix **A** is defined to be the sum of elements on the main diagonal of **A**. The trace of a matrix is the sum of its eigenvalues, and it is invariant with respect to a change of basis.

In [87]:
```
# Calculate L1 Norm
from numpy.linalg import norm
A = np.array([[1, 2, 3],[9, 5, 6]])
x = np.array([1,2,3,4])

l1 = norm(x, 1)
print("L1 norm of Vector x = ",l1) #[(1-0)+(2-0)+(3-0)+(4-0)] = 10

l1 = norm(A, 1)
print("L1 norm of Matrix A = ",l1)#[max((1+9) ,(2+5), (6+3))] = 10
```

```
L1 norm of Vector x =  10.0
L1 norm of Matrix A =  10.0
```

In [91]:
```
# Calculate L2 Norm
A = np.array([[1, 2, 3],[9, 5, 6]])
x = np.array([1,2,3,4])

l2 = norm(x, 2)
print("L2 norm of Vector x = ",l2) #sqrt()

l2 = norm(A, 2)
print("L2 norm of Matrix A = ",l2)
```

```
L2 norm of Vector x =  5.477225575051661
L2 norm of Matrix A =  12.325811872371933
```

4) EigenValues And EigenVectors

- An eigenvector of a square matrix A is a nonzero vector v such that multiplication by A alters only the scale of v and not the direction.

$$Av = \lambda v$$

- The scalar λ is known as the eigenvalue corresponding to this eigenvector.
- The vector Av is the vector v transformed by the matrix A. This transformed vector is a scaled version (scaled by the value λ) of the initial vector v.
- If v is an eigenvector of A, then so is any rescaled vector sv for s $\in$ R, s != 0. Moreover,sv still has the same eigenvalue.

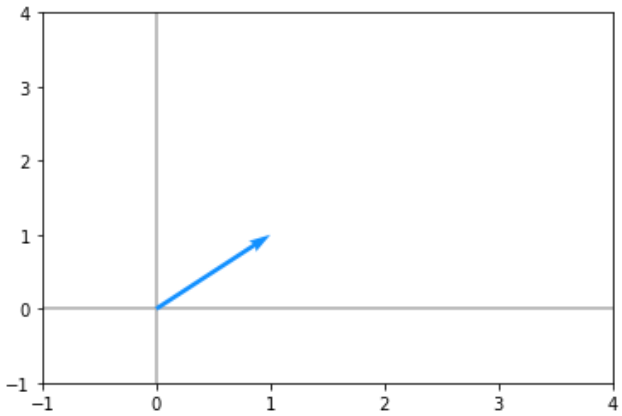
In [2]:
```
# This function is used to plot and visualize transformations
# https://hadrienj.github.io/posts/Deep-Learning-Book-Series-2.7-Eigendecomposition/
import matplotlib.pyplot as plt
def plotVectors(vecs, cols, alpha=1):
    """
    Plot set of vectors.

    Parameters
    ----------
    vecs : array-like
        Coordinates of the vectors to plot. Each vectors is in an array. For
        instance: [[1, 3], [2, 2]] can be used to plot 2 vectors.
    cols : array-like
        Colors of the vectors. For instance: ['red', 'blue'] will display the
        first vector in red and the second in blue.
    alpha : float
        Opacity of vectors

    Returns:

    fig : instance of matplotlib.figure.Figure
        The figure of the vectors
    """
    plt.figure()
    plt.axvline(x=0, color='#A9A9A9', zorder=0)
    plt.axhline(y=0, color='#A9A9A9', zorder=0)

    for i in range(len(vecs)):
        x = np.concatenate([[0,0],vecs[i]])
        plt.quiver([x[0]],
                   [x[1]],
                   [x[2]],
                   [x[3]],
                   angles='xy', scale_units='xy', scale=1, color=cols[i],
                   alpha=alpha)
```

In [114]:
```
A = np.array([[5, 1], [3, 3]])
v = np.array([[1], [1]])
```

In [116]:
```
print("Vector v ")
plotVectors([v.flatten()], cols=['#1190FF'])
plt.ylim(-1, 4)
plt.xlim(-1, 4)
```

```
Vector v
```

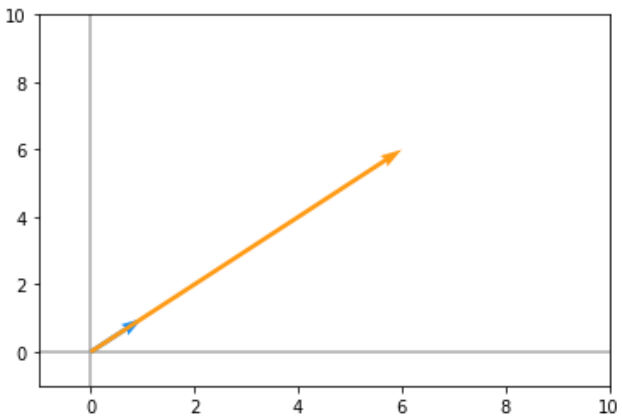
Out[116]: (-1, 4)



In [118]:
```
print("Plotting the Dot Product")
dot = np.dot(A,v)
plotVectors([v.flatten(),dot.flatten()], cols=['#1190FF','#FF9A13'])
plt.ylim(-1, 10)
plt.xlim(-1, 10)
```

```
Plotting the Dot Product
```

Out[118]: (-1, 10)

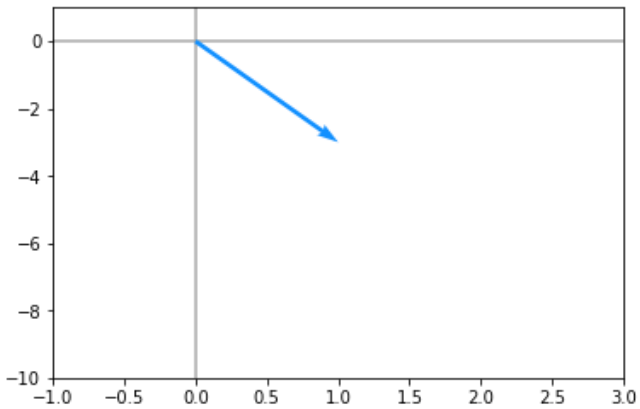


- Their Directions are Same. So we can say that v is an eigen vector of A.

In [123]:
```
v = np.array([[1], [-3]])
print("Vector v ")
plotVectors([v.flatten()], cols=['#1190FF'])
plt.ylim(-10, 1)
plt.xlim(-1, 3)
```

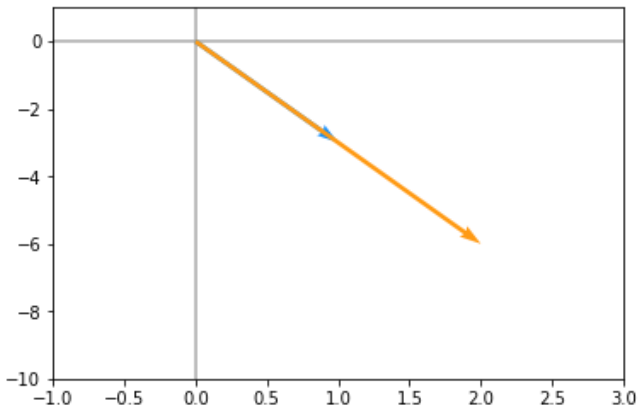
```
Vector v
```

Out[123]: (-1, 3)



In [120]:
```
#This is another eigen vector of A
dot = np.dot(A,v)
plotVectors([v.flatten(),dot.flatten()], cols=['#1190FF','#FF9A13'])
plt.ylim(-10, 1)
plt.xlim(-1, 3)
```

Out[120]: (-1, 3)



- Eigen Vectors are those Vectors(v) when we apply a square matrix A on v, will lie in the same direction as that of v

Finding Eigen Values and Eigen Vectors :

In [139]:
```
eig_val, eig_vec = np.linalg.eig(A)
print("Eigen Values are : ",eig_val)
print("Eigen Vectors are : \n",eig_vec,"\n")
print("Eigen Vector Corresponding to 6 = ",eig_vec[:,0])
print("Eigen Vector Corresponding to 2 = ",eig_vec[:,1])
```

```
Eigen Values are :  [6. 2.]
Eigen Vectors are :
 [[ 0.70710678 -0.31622777]
 [ 0.70710678  0.9486833 ]]

Eigen Vector Corresponding to 6 =  [0.70710678 0.70710678]
Eigen Vector Corresponding to 2 =  [-0.31622777  0.9486833 ]
```

- If [[1],[-3]] is eigen vector of A, then so is [[2],[-6]]

5) Eigen Decomposition :

- Suppose that a matrix A has n linearly independent eigenvectors $\left\{v^1, \ldots, v^n\right\}$ with corresponding eigenvalues $\{\lambda_1, \ldots, \lambda_n\}$.
- We can concatenate all the eigenvectors to form a matrix **V** with one eigenvector per column likewise concatenate all the eigenvalues to form a vector **λ**.
- The eigendecomposition of A is then given by :

$$A = Vdiag(\lambda)V^{-1}$$

- Decomposing a matrix into its corresponding Eigen Values and Eigen Vectors help to analyze properties of the matrix and it helps to understand the behaviour of that matrix.
- Say matrix A is real symmetric matrix, then it can be decomposed as :

$$A = Q\Lambda Q^T$$

- where Q is an orthogonal matrix composed of eigenvectors of A, and Λ is a diagonal matrix.
- Any real symmetric matrix A is guaranteed to have an eigendecomposition, the eigen decomposition may not be unique.
- If any two or more eigenvectors share the same eigenvalue, then any set of orthogonal vectors lying in their span are also eigenvectors with that eigenvalue, and we could equivalently choose a **Q** using those eigenvectors instead.

What does Eigen decomposition tell us?

- The matrix is singular(|A|=0) if and only if any of the eigenvalues are zero.
- The eigen decomposition of a real symmetric matrix can also be used to optimize quadratic expressions of the form f(x) =$x^T Ax$ subject to $||x||_2 = 1$
- Whenever x is equal to an eigenvector of A,f takes on the value of the corresponding eigenvalue.
- The maximum value of f within the constraint region is the maximum eigen value and its minimum value within the constraint region is the minimum eigenvalue.
- A matrix whose eigenvalues are all positive is called **positive definite**.
- A matrix whose eigenvalues are all positive or zero valued is called **positive semidefinite**.
- If all eigenvalues are negative, the matrix is **negative definite**.
- If all eigenvalues are negative or zero valued, it is **negative semidefinite**.
- Positive semidefinite matrices are guarantee that : $\forall x, x^T Ax \geq 0$
- Positive definite matrices additionally guarantee that : $x^T Ax = 0 \Rightarrow x = 0$

In [143]:
```
V = np.array([[1, 1], [1, -3]])
vinv = np.linalg.inv(V)
diag_l = np.diag([6, 2])
A = np.dot(np.dot(V,diag_l),vinv)
print(A)
```

```
[[5. 1.]
 [3. 3.]]
```

6) Singular Value Decomposition

- Factorize a matrix, into **singular vectors** and **singular values**.
- A **singular matrix** is a square matrix which is not invertible. Alternatively, a matrix is singular if and only if it has a determinant of 0.
- The **singular values** are the absolute values of the eigenvalues of a normal matrix A
- The SVD enables us to discover some of the same kind of information as the eigen decomposition reveals;however, the SVD is more generally applicable. Every real matrix has a singular value decomposition, but the same is not true of the eigenvalue decomposition.
- The singular value decomposition is similar to Eigen Decomposition except this time we will write **A** as a product of three matrices:

$$A = UDV^T$$

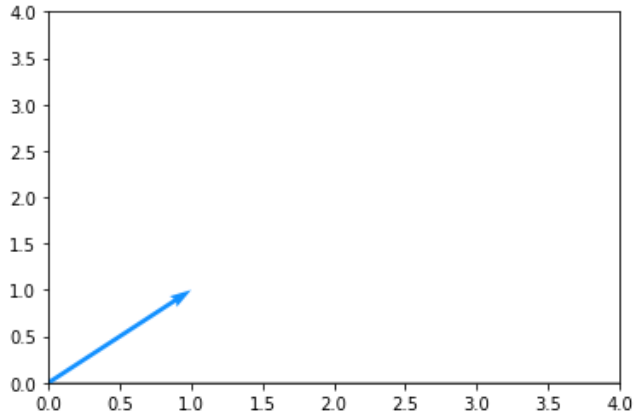
  .
- U and V are orthogonal matrices ($U^T = U^{-1} and V^T = V^{-1}$)
- D is a diagonal matrix (all 0 except the diagonal) and need not be square.
- The columns of U are called the left-singular vectors of A while the columns of V are the right-singular vectors of A. The values along the diagonal of D are the singular values of A.
- Suppose that **A** is an m ×n matrix. Then **U** is defined to be an m × m matrix, **D** to be an m × n matrix, and **V** to be an n × n matrix

Intuition:

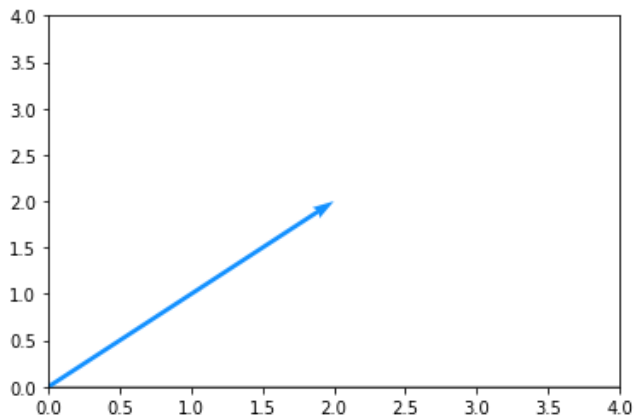
- **A** is a matrix that can be seen as a linear transformation. This transformation can be decomposed in three sub-transformations: 1. rotation, 2. re-scaling, 3. rotation. These three steps correspond to the three matrices **U**, **D**, and **V**.

In [173]:
```
print("Assume that We have the following Vector(v) :")
v = np.array([[1], [1]])
print(v)
plotVectors([v.flatten()], cols=['#1190FF'])
plt.ylim(0, 4)
plt.xlim(0, 4)
plt.show()
#Applying Transformation
A = np.array([[2, 0],[0, 2]])
print("Assume we have the Matrix A:")
print(A)
print("Applying Matrix A to vector v, we get")
Adotv = np.dot(A,v)
plotVectors([Adotv.flatten()], cols=['#1190FF'])
plt.ylim(0, 4)
plt.xlim(0, 4)
plt.show()
```

```
Assume that We have the following Vector(v) :
[[1]
 [1]]
```



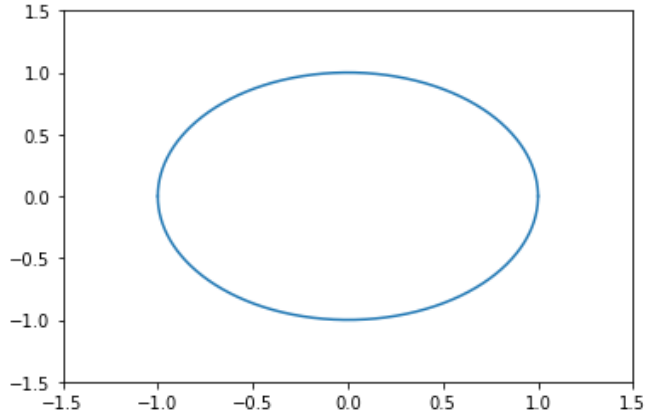

```
Assume we have the Matrix A:
[[2 0]
 [0 2]]
Applying Matrix A to vector v, we get
```



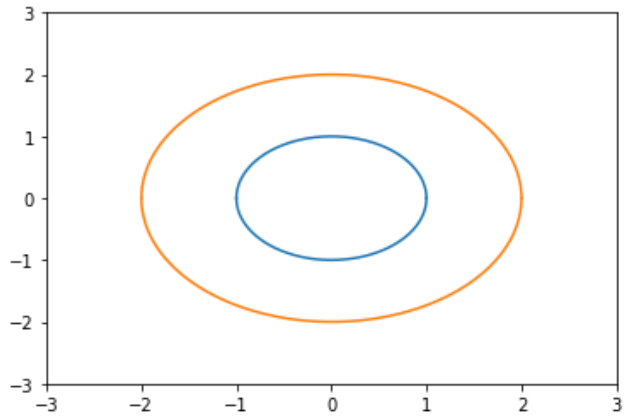
- Doubled each coordinate of our vector

In [213]:
```
import seaborn as sns
#Assume we have a circle now and see how the matrix A transforms the Circle
#A = np.array([[2, 0],[0, 2]])
print("Assume we have the following circle")
x = np.linspace(-1, 1, 100000)
y = np.sqrt(1-(x**2))
plt.plot(x, y, sns.color_palette().as_hex()[0])
plt.plot(x, -y, sns.color_palette().as_hex()[0])
plt.xlim(-1.5, 1.5)
plt.ylim(-1.5, 1.5)
plt.show()
print("Now to this circle we apply the the matrix A")
comb = np.row_stack((x,y)) #Combine both x and y column-wise
xx = np.dot(A,comb) #Take the dot product

#Plotting:
#plot the old circle:
plt.plot(x, y, sns.color_palette().as_hex()[0])
plt.plot(x, -y, sns.color_palette().as_hex()[0])
#plot the new circle:
plt.plot(xx[0], xx[1], sns.color_palette().as_hex()[1])
plt.plot(xx[0], -xx[1], sns.color_palette().as_hex()[1])
plt.xlim(-3, 3)
plt.ylim(-3, 3)
plt.show()

#New matrix A :
A = np.array([[3, 0],[0, 2]])
print("New Array A : ")
print(A)
print("Now we apply the the matrix A to the first circle")
xx = np.dot(A,comb) #Take the dot product of combined matrix to A
#plot the old circle:
plt.plot(x, y, sns.color_palette().as_hex()[0])
plt.plot(x, -y, sns.color_palette().as_hex()[0])
#plot the new circle:
plt.plot(xx[0], xx[1], sns.color_palette().as_hex()[1])
plt.plot(xx[0], -xx[1], sns.color_palette().as_hex()[1])
plt.xlim(-4, 4)
plt.ylim(-4, 4)
plt.show()
```

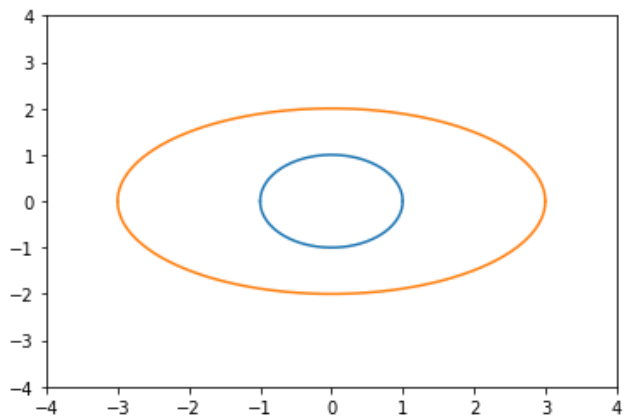
```
Assume we have the following circle
```




```
Now to this circle we apply the the matrix A
```




```
New Array A :
[[3 0]
 [0 2]]
Now we apply the the matrix A to the first circle
```

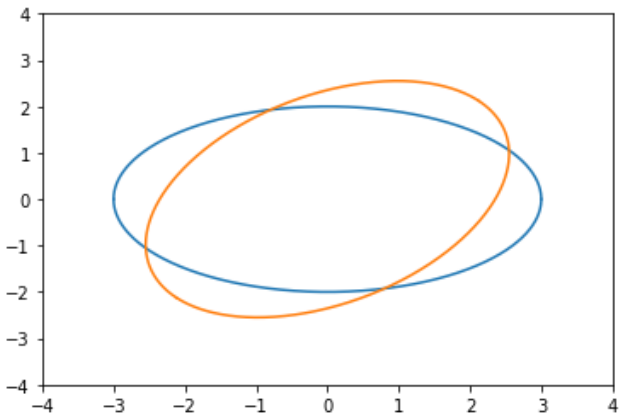


- The transformation associated with Diagonal matrix imply rescaling of each coordinates without rotation.

  ## Rotation :

In [225]:
```
#Rotation Matrix : We rotate each axis by 45 degrees
A = np.array([[np.cos(np.radians(45)) , -np.sin(np.radians(45))],
     [np.sin(np.radians(45)), np.cos(np.radians(45))]])
A_neg = np.array([[np.cos(np.radians(45)) , np.sin(np.radians(45))],
     [np.sin(np.radians(45)), -np.cos(np.radians(45))]])

print("Now we apply the the Rotation matrix A to the rescaled circle")
#apply Transformation
new_pos = np.dot(A,xx)
new_neg = np.dot(A_neg,xx)
#Plot old circle
plt.plot(xx[0], xx[1], sns.color_palette().as_hex()[0])
plt.plot(xx[0], -xx[1], sns.color_palette().as_hex()[0])

#plot new circle
plt.plot(new_pos[0], new_pos[1], sns.color_palette().as_hex()[1])
plt.plot(new_neg[0], new_neg[1], sns.color_palette().as_hex()[1])

plt.xlim(-4, 4)
plt.ylim(-4, 4)
plt.show()
```

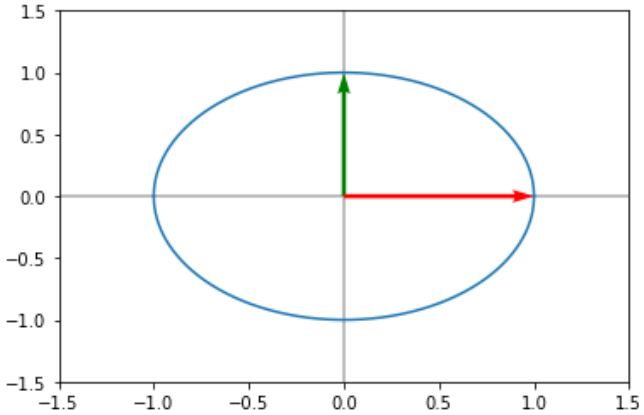
```
Now we apply the the Rotation matrix A to the rescaled circle
```



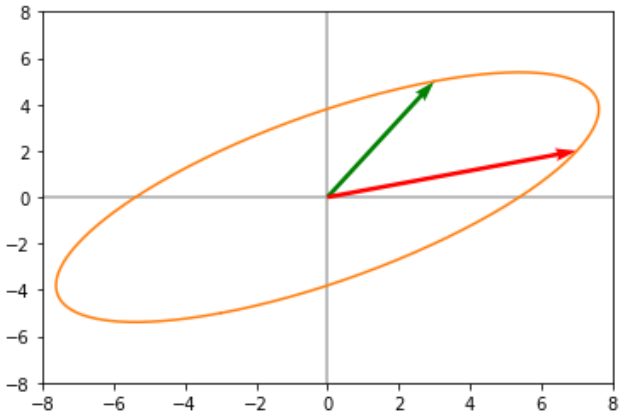
- The circle rotated by an angle of 45 degrees.
- Note: SVD can even be applied on a non-square matrix as well.
- The SVD can be seen as the decomposition of one complex transformation in 3 simpler transformations (a rotation, a scaling and another rotation)

In [36]:
```
#Our new Transformation Matrix:
print("New Transformation Matrix : ")
A = np.array([[3 , 7],[5, 2]])
print(A)
print("Our Original Unit Circle")
x = np.linspace(-1, 1, 100000)
y = np.sqrt(1-(x**2))


plotVectors([[0,1], [1,0]], cols=['green', 'red'])


plt.plot(x, y, sns.color_palette().as_hex()[0])
plt.plot(x, -y, sns.color_palette().as_hex()[0])
plt.xlim(-1.5, 1.5)
plt.ylim(-1.5, 1.5)


plt.show()
print("Applying A to the unit circle : ")
comb = np.row_stack((x,y))
trans_pos = np.dot(A,comb)
trans_neg = np.dot(np.array([[3 , -7],[5, -2]]),comb)
#Plot original unit circle

u1 = [A[0,0],A[1,0]]
v1 = [A[0,1],A[1,1]]

plotVectors([u1, v1], cols=['green', 'red'])
#plt.plot(x, y, sns.color_palette().as_hex()[0])
#plt.plot(x, -y, sns.color_palette().as_hex()[0])
#Plot transformed circle

plt.plot(trans_pos[0], trans_pos[1], sns.color_palette().as_hex()[1])
plt.plot(trans_neg[0], trans_neg[1], sns.color_palette().as_hex()[1])
plt.xlim(-8, 8)
plt.ylim(-8, 8)
plt.show()
```

```
New Transformation Matrix :
[[3 7]
 [5 2]]
Our Original Unit Circle
```




```
Applying A to the unit circle :
```



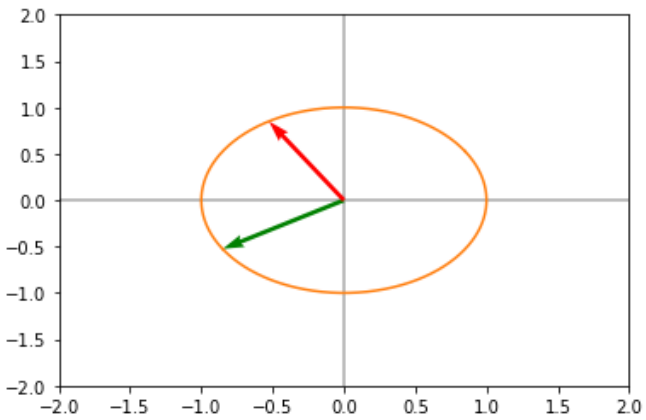
Now Let's find the SVD of A

In [8]:
```
U, D, V = np.linalg.svd(A)
print("Matrix U : \n",U)
print("D : \n",D)
print("Matrix V : \n",V)
```

```
Matrix U :
 [[-0.85065081 -0.52573111]
 [-0.52573111  0.85065081]]
D :
 [8.71337969 3.32821489]
Matrix V :
 [[-0.59455781 -0.80405286]
 [ 0.80405286 -0.59455781]]
```

In [30]:
```
#Applying U to the unit circle(First Rotation)
print("Apply U to unit circle : First Rotation ")
U_pos = np.dot(U,comb)
U_neg = np.dot(np.array([[U[0,0] ,-U[0,1]], [U[1,0],-U[1,1]]]),comb)

u1 = [U[0,0],U[1,0]]
v1 = [U[0,1],U[1,1]]

plotVectors([u1, v1], cols=['green', 'red'])

plt.plot(U_pos[0], U_pos[1], sns.color_palette().as_hex()[1])
plt.plot(U_neg[0], U_neg[1], sns.color_palette().as_hex()[1])
plt.xlim(-2, 2)
plt.ylim(-2, 2)

plt.show()
print("Transformation can be Observed very clearly here")
```

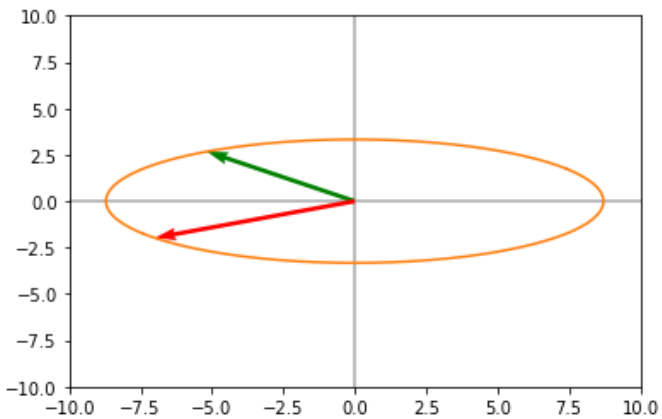
```
Apply U to unit circle : First Rotation
```




```
Transformation can be Observed very clearly here
```

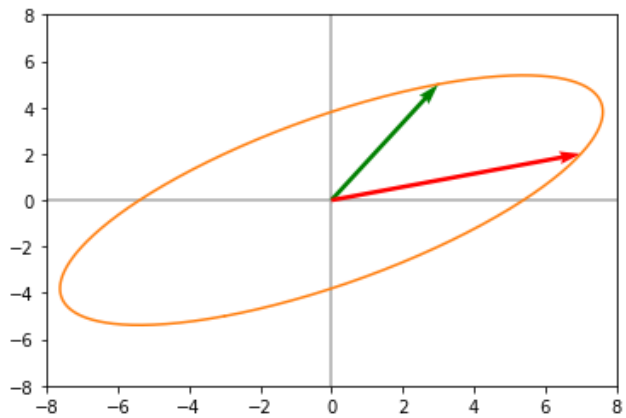
In [31]:
```
print("Applying Scaling")
d_pos = np.dot(np.diag(D),U_pos)
d_neg = np.dot(np.diag(D),U_neg)

d_vec = np.diag(D).dot(V)
u1 = [d_vec[0,0],d_vec[1,0]]
v1 = [d_vec[0,1],d_vec[1,1]]

plotVectors([u1, v1], cols=['green', 'red'])

plt.plot(d_pos[0], d_pos[1], sns.color_palette().as_hex()[1])
plt.plot(d_neg[0], d_neg[1], sns.color_palette().as_hex()[1])
plt.xlim(-10, 10)
plt.ylim(-10, 10)
plt.show()
```

```
Applying Scaling
```



In [32]:
```
print("Applying the Last Rotation Matrix")
fin = np.dot(U,np.diag(D))
fin1 = np.dot(fin,V)

d_vec = U.dot(np.diag(D)).dot(V)
u1 = [d_vec[0,0],d_vec[1,0]]
v1 = [d_vec[0,1],d_vec[1,1]]

plotVectors([u1, v1], cols=['green', 'red'])



V_pos = np.dot(fin1,comb)
V_neg = np.dot(np.array([[fin1[0,0] ,-fin1[0,1]], [fin1[1,0],-fin1[1,1]]]),comb)

plt.plot(V_pos[0], V_pos[1], sns.color_palette().as_hex()[1])
plt.plot(V_neg[0], V_neg[1], sns.color_palette().as_hex()[1])
plt.xlim(-8, 8)
plt.ylim(-8, 8)
plt.show()
```

```
Applying the Last Rotation Matrix
```



- This is exactly same as the previous one. This proves SVD

- Singular Values are ordered in descending order. They correspond to a new set of features (that are a linear combination of the original features) with the first feature explaining most of the variance.

How to find the sub-transformations

- U corresponds to the eigenvectors of AA^T
- V corresponds to the eigenvectors of A^TA
- D corresponds to the eigenvalues AA^T or A^TA which are the same.

Proof :

In [287]:
```
#Using SVD from linalg
A = np.array([[4, 2], [6, 4], [7, 3]])
U, D, V = np.linalg.svd(A)
print("U : \n",U)
print("V : \n",V)
print("D : \n",np.diag(D))
```

```
U :
 [[-0.39353446  0.10861537 -0.91287093]
 [-0.63149881 -0.75357542  0.18257419]
 [-0.66808673  0.64832614  0.36514837]]
V :
 [[-0.88370896 -0.46803684]
 [ 0.46803684 -0.88370896]]
D :
 [[11.36090991  0.        ]
 [ 0.          0.96422305]]
```

- Manually Compute U,V,D

In [293]:
```
u_part = np.dot(A,A.T)
eig_vec = np.linalg.eig(u_part)[1]
eig_vec
```

Out[293]:
```
array([[ 0.39353446,  0.91287093,  0.10861537],
       [ 0.63149881, -0.18257419, -0.75357542],
       [ 0.66808673, -0.36514837,  0.64832614]])
```

In [295]:
```
v_part = np.dot(A.T,A)
eig_vec = np.linalg.eig(v_part)[1]
eig_vec
```

Out[295]:
```
array([[ 0.88370896, -0.46803684],
       [ 0.46803684,  0.88370896]])
```

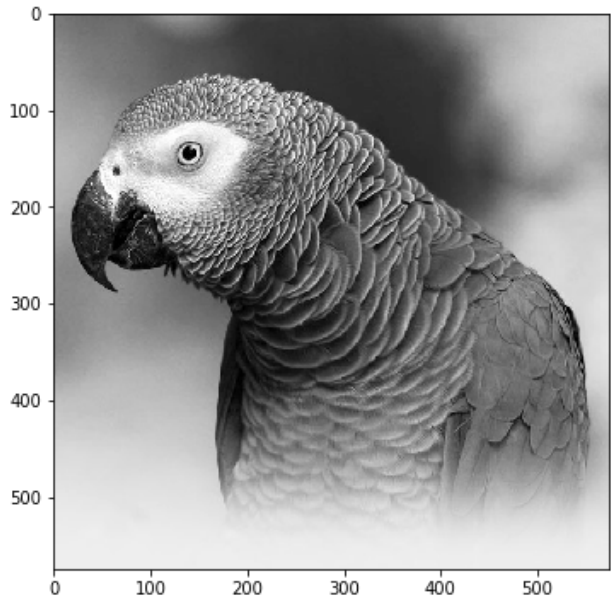
In [299]:
```
#D will be square root of eigen values of A.T.dot(A)
v_part = np.dot(A.T,A)
eig_val = np.linalg.eig(v_part)[0]
np.diag(np.sqrt(eig_val))
```

Out[299]:
```
array([[11.36090991,  0.        ],
       [ 0.        ,  0.96422305]])
```

6.1) SVD For Image Compression

In [303]:
```
# We will Load an image and then perform Dimensionality Reduction using SVD:
import cv2
img = cv2.imread("svd_dim_red.jpg",0)
print(img.shape)
plt.figure(figsize=(9, 6))
plt.imshow(img, cmap='gray')
plt.show()
```

```
(575, 575)
```

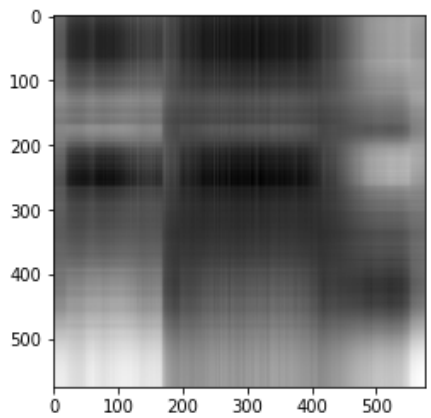


In [304]:
```
# Apply SVD to this image
U, D, V = np.linalg.svd(img)
print(U.shape)
print(D.shape)
print(V.shape)
```

```
(575, 575)
(575,)
(575, 575)
```

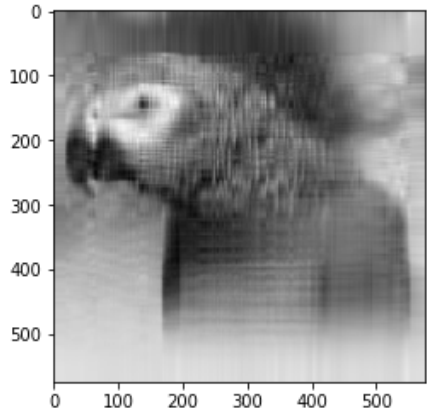
- The singular vectors and singular values are ordered with the first ones corresponding to the more variance explained. For this reason, using just the first few singular vectors and singular values will provide the reconstruction of the principal elements of the image.

In [305]:
```
#Visualize the image using 2 singular values
reconstimg = np.matrix(U[:, :2]) * np.diag(D[:2]) * np.matrix(V[:2, :])
plt.imshow(reconstimg, cmap='gray')
plt.show()
```

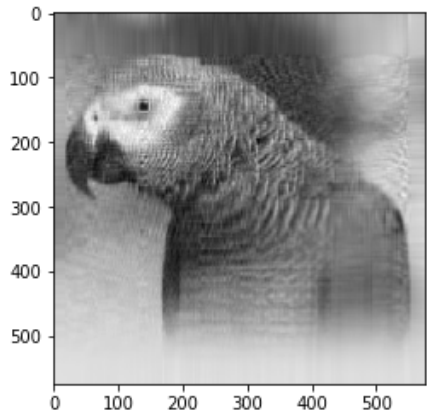


In [309]:
```
#We will now draw the reconstruction using different number of singular values.

components = [10,20,30,40,50,60]
for i in components:
    print("Number of components = ", i)
    reconstimg = np.matrix(U[:, :i]) * np.diag(D[:i]) * np.matrix(V[:i, :])
    plt.imshow(reconstimg, cmap='gray')
    plt.show()
```

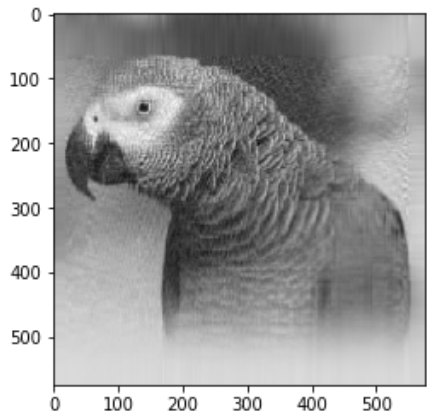
```
Number of components =  10
```



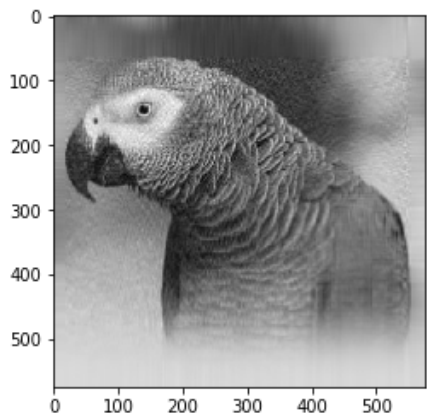

```
Number of components =  20
```



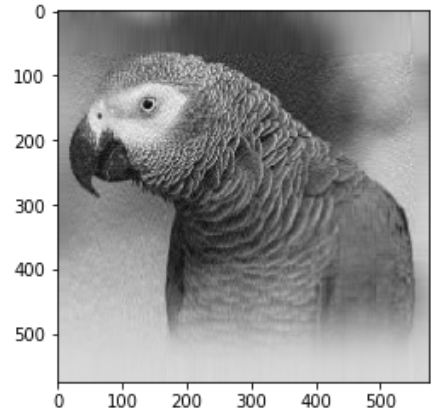

```
Number of components =  30
```



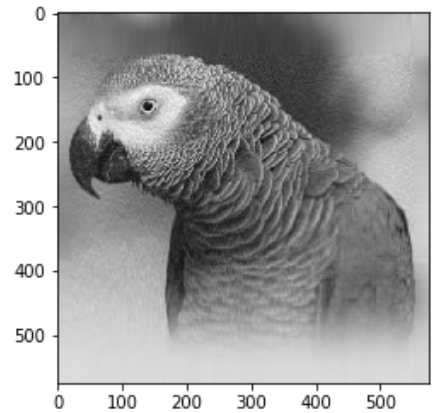

```
Number of components =  40
```




```
Number of components =  50
```

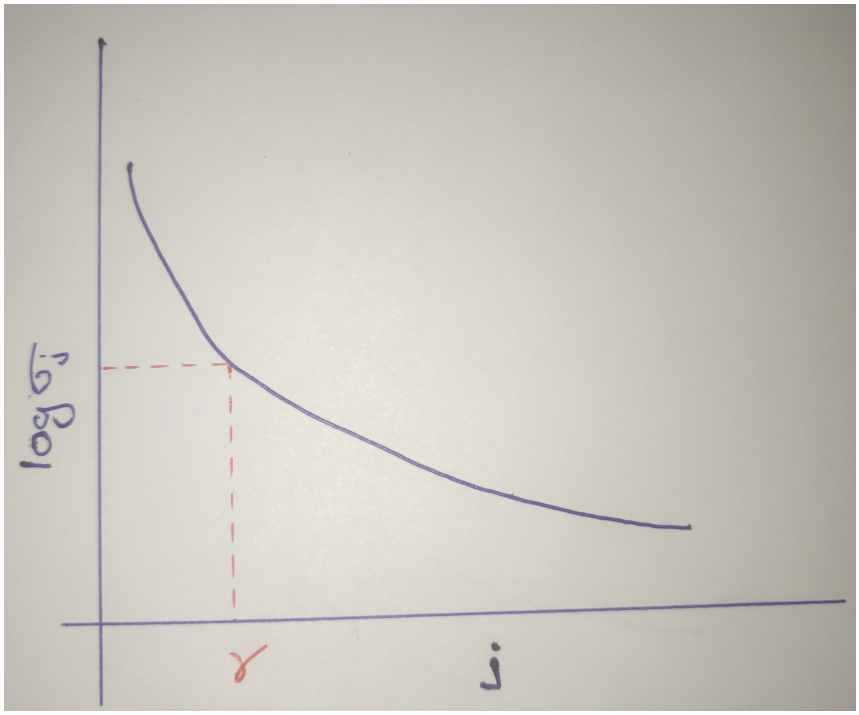



```
Number of components =  60
```



6.2) Truncated SVD

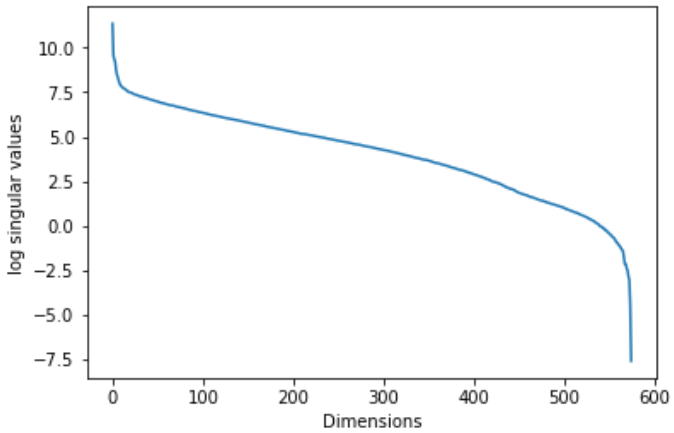
- From SVD we know that any matrix A can be decomposed as product of 3 matrices:
$$A = UDV^T$$
.
- Now we can choose to keep only the first r columns of **U**, r columns of **V** and $r \times r$ submatix of **D**.
- **How to choose r?**
- If r is large => more complex model and more accurate model.
- If r is small => less complex model and less accurate model.
- So we need to fina a sweet spot where we get more information in **A**
- Plot the log of the singular values(diagonal values σ) and number of components and we will expect to see an elbow in the graph and use that to pick the value for r.



- But it does not work out unless you get a clear drop-off in the singular values.
- Plotting our Image Data's singular values

In [344]:
```
#Plot of our Singular Values:
plt.plot(np.log(D))
plt.xlabel("Dimensions")
plt.ylabel("log singular values")
```

Out[344]: Text(0,0.5,'log singular values')



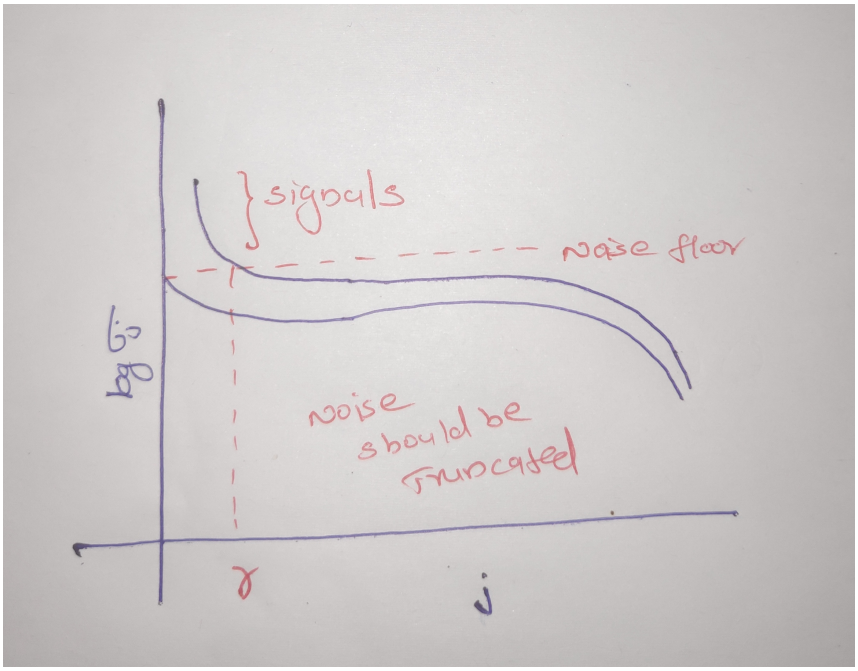
- What do we do in case of above situation :
- We can use the ideas from this paper : https://arxiv.org/pdf/1305.5870.pdf (https://arxiv.org/pdf/1305.5870.pdf)

- We can write the Data Matrix **A** as :

$$A = A_{true} + \gamma A_{noise}$$

- The fundamental Assumption is that :

$$A_{noise} \approx N(0, 1)$$



- If Data has low rank structure(ie we use a cost function to measure the fit between the given data and it's approximation) and a Gaussian Noise added to it, We find the first singular value which is larger than the largest singular value of the noise matrix.

- We keep all those singular values and truncate everything that is smaller than Noise floor.

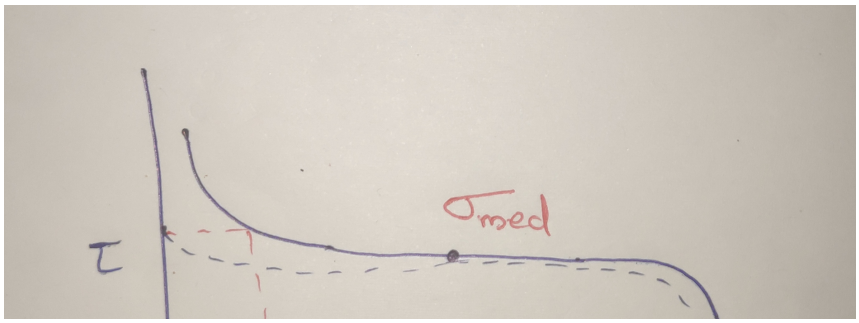
Case 1:(Best Case Scenario)

- **A** is a Square Matrix and γ is known.
- Here we truncate all $\sigma < \tau$ (Threshold)

$$\tau = \frac{4}{\sqrt{3}}\gamma\sqrt{n}$$

Case 2:

- **A** is a Non-square Matrix ($m \neq n$) and γ is not known.

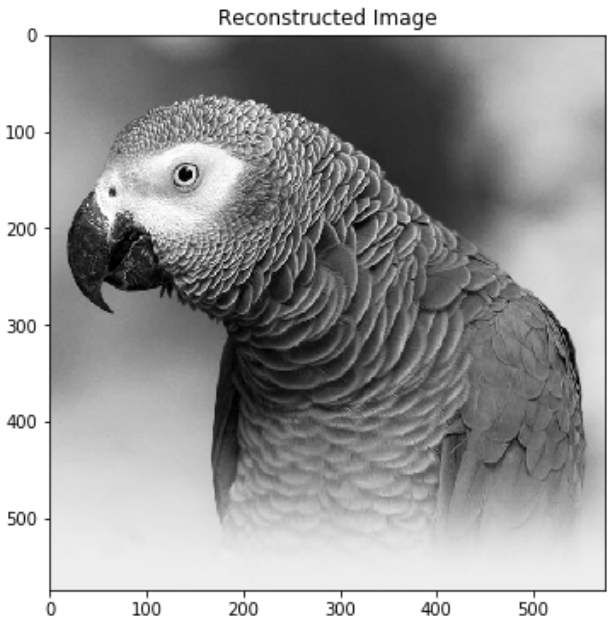


In [408]:
```
sig_med = np.median(D)
beta = img.shape[0]/img.shape[1]
omega = 0.56*beta**3 - 0.95*beta**2 + 1.82*beta + 1.43
tou = omega*sig_med
print("Value of Tou = ",tou)
print("Now we need to eliminate all singular values < ",tou)
```

```
Value of Tou =  233.1276285790134
Now we need to eliminate all singular values <  233.1276285790134
```

In [411]:
```
new_sing_vals = D[D<tou]
print("So we need only",new_sing_vals.shape[0]," components to explain most of the va
riance")
print("Reconstructing The Image")
reconstimg = np.matrix(U[:, :new_sing_vals.shape[0]]) * np.diag(D[:new_sing_vals.shap
e[0]]) * np.matrix(V[:new_sing_vals.shape[0], :])
plt.figure(figsize=(9, 6))
plt.imshow(reconstimg, cmap='gray')
plt.title("Reconstructed Image")
plt.show()
```

```
So we need only 392  components to explain most of the variance
Reconstructing The Image
```



6.3) Truncated SVD With Sklearn :

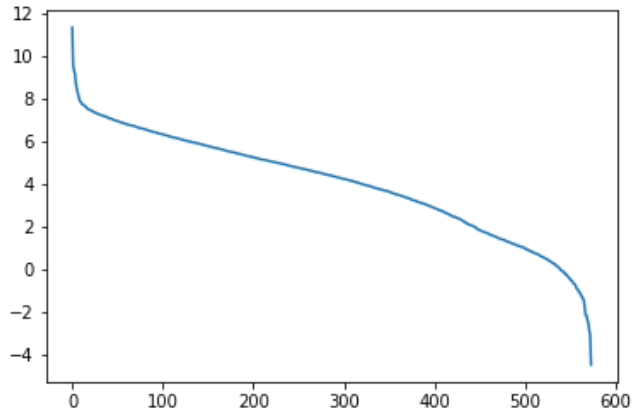
In [400]:
```
from sklearn.decomposition import TruncatedSVD

print("Data Shape = ",img.shape)
svd = TruncatedSVD(n_components=574, n_iter=7, random_state=42)
svd.fit(img)
```

```
Data Shape =  (575, 575)
```

Out[400]:
```
TruncatedSVD(algorithm='randomized', n_components=574, n_iter=7,
       random_state=42, tol=0.0)
```

In [401]:
```
plt.plot(np.log(svd.singular_values_))
```

Out[401]:
```
[<matplotlib.lines.Line2D at 0x11d39c940>]
```



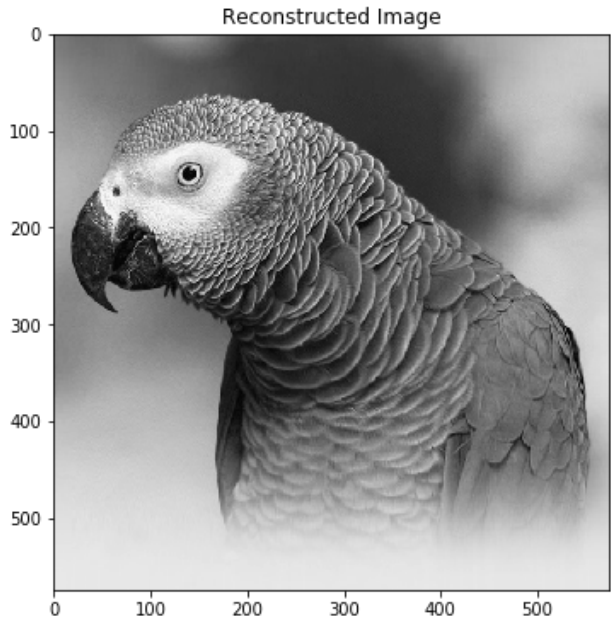
In [402]:
```
sig_med = np.median(svd.singular_values_)
beta = img.shape[0]/img.shape[1]
omega = 0.56*beta**3 - 0.95*beta**2 + 1.82*beta + 1.43
tou = omega*sig_med
print("Value of Tou = ",tou)
print("Now we need to eliminate all singular values < ",tou)
```

```
Value of Tou =  234.55504407897726
Now we need to eliminate all singular values <  234.55504407897726
```

In [403]:
```
new_sing_vals = svd.singular_values_[svd.singular_values_<tou]
print("So we need only",new_sing_vals.shape[0]," components to explain most of the va
riance")
print("Apply SVD with new Components")
svd = TruncatedSVD(n_components=new_sing_vals.shape[0], n_iter=7, random_state=42)
out_img = svd.fit_transform(img)
print("Transformed Image Shape = ",out_img.shape)
reconst_img = svd.inverse_transform(out_img)
print("Reconstructed Image Shape = ",reconst_img.shape)
```

```
So we need only 392  components to explain most of the variance
Apply SVD with new Components
Transformed Image Shape =  (575, 392)
Reconstructed Image Shape =  (575, 575)
```

In [404]:
```
plt.figure(figsize=(9, 6))
plt.imshow(fin_img, cmap='gray')
plt.title("Reconstructed Image")
plt.show()
```



7) Principal Components Analysis (PCA)

- Suppose we have m data points $\{x^1 \dots x^m\}$ in R^n and we wish to apply a lossy compression to these points so that we can store these points in a lesser memory but may loose some precision. So the objective is to loose as little as precision as possible.
- So we convert these points to a lower dimensional version. For each point $x^{(i)} \in R^n$ we will find a corresponding code vector $c^{(i)} \in R^l$. If l is less than n, then it requires less space for storage.
- We need to find an encoding function that will produce the encoded form of the input $f(x) = c$ and a decoding function that will produce the reconstructed input given the encoded form $x \approx g(f(x))$.
- The encoding function $f(x)$ transforms x into c and the decoding function transforms back c into an approximation of x .

Constraints :

1) The decoding function has to be a simple matrix multiplication:

$$g(c) = Dc$$

When we apply the matrix D to the Dataset of new coordinate system, we will should get back the dataset in the initial coordinate system.

2) The columns of D must be orthogonal(D is semi-orthogonal D will be an Orthogonal matrix when n=l)

3) The columns of D must have unit norm

7.1) Finding the encoding function

- We know $g(c) = Dc$. We will find the encoding function from the decoding function.
- We want to minimize the error between the decoded data point and the actual data point. That is we want to reduce the distance between x and g(c). We can measure this distance using the L^2 Norm.
- We need to minimize the following:

$$\|x - g(c))\|_2^2$$

- We will use the Squared L^2 norm because both are minimized using the same value for c
- Let c^* be the optimal c. Mathematically we can write it as :

$$c^* = \|x - g(c))\|_2^2$$

- But Squared L^2 norm can be expressed as :

$$\|x\|_2^2 = x^T x$$

- Applying the same formula here we get :

$$(x - g(c))^T (x - g(c))$$

$$(x^T - g(c)^T)(x - g(c))$$

- By the distributive property, we get:

$$x^T x - x^T g(c) - g(c)^T x + g(c)^T g(c)$$

- The commutative property that $x^T y = y^T x$. Since the result of $g(c)^T x$ is a scalar, we have $g(c)^T x = x^T g(c)$. So the equation becomes:

$$x^T x - x^T g(c) - x^T g(c) + g(c)^T g(c)$$

$$= x^T x - 2x^T g(c) + g(c)^T g(c)$$

- The first term $x^T x$ does not depends on c and since we want to minimize the function according to c we can just get off this term :

$$c^* = argmin_c - 2x^T g(c) + g(c)^T g(c)$$

- Substituting $g(c) = Dc$:

$$c^* = argmin_c - 2x^T Dc + (Dc)^T Dc$$

- But $(Dc)^T = c^T D^T$. So the equation now becomes :

$$c^* = argmin_c - 2x^T Dc + c^T D^T Dc$$

- $D^T D = I_l$ because D is orthogonal (actually, it is semi-orthogonal if n≠l) and have unit norm columns :

$$c^* = argmin_c - 2x^T Dc + c^T I_l c$$

$$c^* = argmin_c - 2x^T Dc + c^T c$$

- Now we can minimize this function using Gradient Descent. The main idea is that the sign of the derivative of the function at a specific value of x tells you if you need to increase or decrease x to reach the minimum. When the slope is near 0, the minimum should have been reached.

$$\nabla_c(-2x^T Dc + c^T c) = 0$$

$$-2x^T D + 2c = 0$$

$$c = x^T D$$

- We want c to be a column vector of shape (l, 1), so we need to transpose $x^T D$.

$$(x^T D)^T = D^T x$$

- So we get :

$$c = D^T x$$

- To encode a vector, we apply the encoder function

$$f(x) = D^T x$$

- Reconstruction :

$$r(x) = g(f(x)) = DD^T x = x$$

7.2) Choose the encoding matrix D

- Purpose of the PCA is to change the coordinate system in order to maximize the variance along the first dimensions of the projected space.
- Maximizing the variance corresponds to minimizing the error of the reconstruction.
- Since we will use the same matrix **D** to decode all the points, we can no longer consider the points in isolation. Instead, we must minimize the Frobenius norm of the matrixof errors computed over all dimensions and all points:
- Forbeus Norm Formula :

$$\|A\|_F = \sqrt{\sum_{i,j} A_{i,j}^2}$$

- It is like if you unroll the matrix to end up with a one dimensional vector and that you take the L^2 norm of this vector.
- Using This formula we get :

$$D^* = argmin_D \sqrt{\sum_{i,j} \left(x_j^{(i)} - r\left(x^{(i)}\right)_j \right)^2}$$

- **Constraint:** $D^T D = I_l$
- We will start to find only the first principal component (PC). For that reason, we will have l=1. So the matrix D will have the shape (n×1). Since it is a column vector, we can call it **d**

$$d^* = argmin_d \sum_i \left\| x^{(i)} - r(x^{(i)}) \right\|_2^2$$

$$r(x) = DD^T x$$

- Since we are looking only for the first PC:

$$r(x) = dd^T x$$

- Plugging this r(x) into the equation, we get:

$$d^* = argmin_d \sum_i \left\| x^{(i)} - dd^T x^{(i)} \right\|_2^2$$

$$\|d\|_2 = 1$$

- We want $x(i)^T$ instead of $x^{(i)}$ in our expression of d^* . So taking transpose we get :

$$d^* = argmin_d \sum_i \left\| \left(x^{(i)} - dd^T x^{(i)} \right)^T \right\|_2^2$$

$$= argmin_d \sum_i \left\| (x^{(i)})^T - (x^{(i)})^T dd^T \right\|_2^2$$

$$d^* = argmin_d \left\| X - Xdd^T \right\|_F^2$$

$$d^T d = 1$$

- Using the Trace Operator to simplify the above equation

$$||A||_F = \sqrt{Tr(AA^T)}$$

- Now using this in our equation for d^*, we get :

$$d^* = argmin_d Tr((X - Xdd^T)(X - Xdd^T)^T)$$

- Now $Tr(AB) = Tr(BA)$. Using this we get :

$$d^* = argmin_d Tr((X - Xdd^T)^T (X - Xdd^T))$$

$$= argmin_d Tr((X^T - (Xdd^T)^T)(X - Xdd^T))$$

$$(Xdd^T)^T = (d^T)^T d^T X^T = dd^T X^T$$

- Plugging this into the equation, we will get :

$$d^* = argmin_d Tr(X^T - dd^T X^T)(X - Xdd^T))$$

$$= argmin_d Tr(X^T X - X^T Xdd^T - dd^T X^T X + dd^T X^T Xdd^T)$$

$$= argmin_d Tr(X^T X) - Tr(X^T Xdd^T) - Tr(dd^T X^T X) + Tr(dd^T X^T Xdd^T)$$

- We need to minimize for d, so we can remove all the terms that do not contain d :

$$d^* = argmin_d - Tr(X^T Xdd^T) - Tr(dd^T X^T X) + Tr(dd^T X^T Xdd^T)$$

- Due to cyclic property of Trace : $Tr(X^T Xdd^T) = Tr(dd^T X^T X)$

$$d^* = argmin_d - 2Tr(X^T Xdd^T) + Tr(dd^T X^T Xdd^T)$$

$$d^* = argmin_d - 2Tr(X^T Xdd^T) + Tr(X^T Xdd^T dd^T)$$

- Now we know $d^T d = 1$

$$d^* = argmin_d - 2Tr(X^T Xdd^T) + Tr(X^T Xdd^T)$$

- subject to $d^T d = 1$

$$= argmin_d - Tr(X^T Xdd^T)$$

- subject to $d^T d = 1$

$$argmax_d Tr(X^T Xdd^T)$$

- subject to $d^T d = 1$

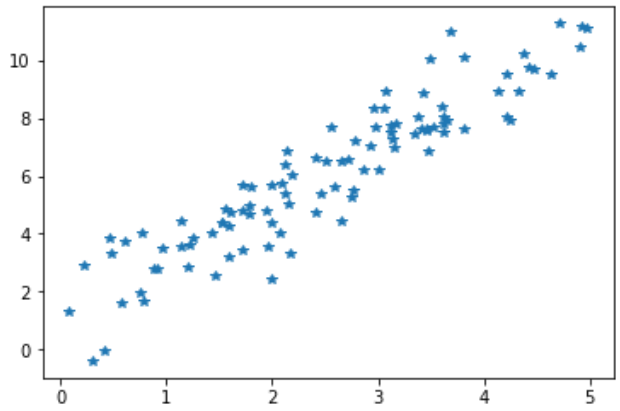
$$d^* = argmax_d Tr(d^T X^T Xd)$$

- subject to $d^T d = 1$
- We can use Eigen Decomposition to solve the equation. The optimal d is given by the the eigen vector of $X^T X$ corresponding to largest Eigen value.
- The matrix $X^T X$ is called the Covariance Matrix when we center the data around 0.
- The covariance matrix is a n by n matrix. Its diagonal is the variance of the corresponding dimensions.
- Other cells are the Covariance between the two corresponding dimensions which tells us the amount of redundancy.
- This means that the largest covariance we have between two dimensions the more redundancy exists between these dimensions. That means if variance is high, then we get small errors.
- To maximize the variance and minimize the covariance (in order to decorrelate the dimensions) means that the ideal covariance matrix is a diagonal matrix (non-zero values in the diagonal only).
- The diagonalization of the covariance matrix will give us the optimal solution.

In [310]:
```
#Create a Random Array:
np.random.seed(123)
x = 5*np.random.rand(100)
y = 2*x + 1 + np.random.randn(100)

x = x.reshape(100, 1)
y = y.reshape(100, 1)

X = np.hstack([x, y])
X.shape
```

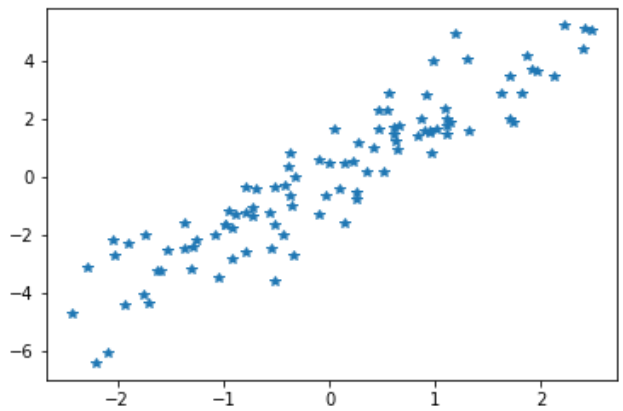
Out[310]: (100, 2)

In [311]:
```
#Let's plot the two dimensions of x :
plt.plot(X[:,0], X[:,1], '*')
plt.show()
```



- When data is highly correlated, then dimensions are redundant and we can predict one from another without losing much information.

In [315]:
```
# Now we will center the data :

X_center =X - np.mean(X, axis = 0)

plt.plot(X_center[:,0], X_center[:,1], '*')
plt.show()
```



- We need to find the Principal Component now :
- Since this is 2-d we can have only 2 components. So we need to find d which maximizes the function :

$$d^* = argmax_d Tr(d^T X^T X d)$$

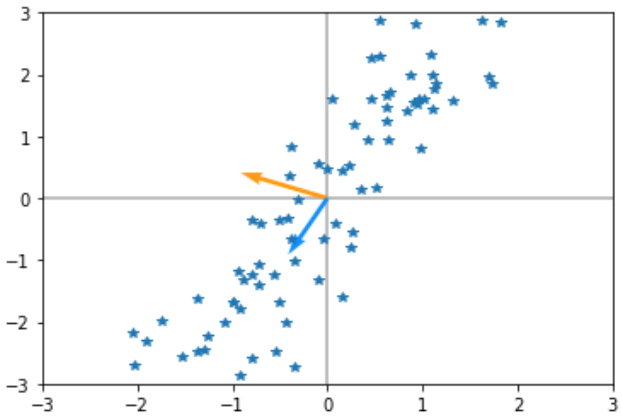
- To find d we need to calculate the eigenvectors of $X^T X$

In [329]:
```
eigVals,  eigVecs = np.linalg.eig(X_center.T.dot(X_center))
print("Eigen Vectors = \n", eigVecs)
print("Eigen Values = \n",np.diag(eigVals))
```

```
Eigen Vectors =
 [[-0.91116273 -0.41204669]
 [ 0.41204669 -0.91116273]]
Eigen Values =
 [[ 18.04730409   0.        ]
 [  0.         798.35242844]]
```

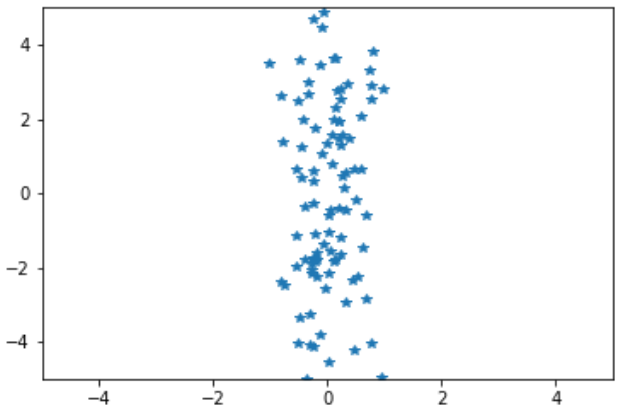
- The eigen vector associated with the larger eigenvalue tells us the direction associated with the larger variance in our data.

In [330]:
```
orange = '#FF9A13'
blue = '#1190FF'
plotVectors(eigVecs.T, [orange, blue])
plt.plot(X_center[:,0], X_center[:,1], '*')
plt.xlim(-3, 3)
plt.ylim(-3, 3)
plt.show()
```



- If we project the data points on the line corresponding to the blue vector direction, we will end up with the largest variance. This vector has the direction that maximizes variance of projected data.
- Now that we have found the matrix d we will use the encoding function to rotate the data. **The goal of the rotation is to end up with a new coordinate system where data is uncorrelated and thus where the basis axes gather all the variance. It is then possible to keep only few axes: this is the purpose of dimensionality reduction.**
- Encoding function $c = D^T x$. D is the matrix containing the eigenvectors

In [332]:
```
X_new = eigVecs.T.dot(X_center.T)
plt.plot(X_new[0, :], X_new[1, :], '*')
plt.xlim(-5, 5)
plt.ylim(-5, 5)
plt.show()
```



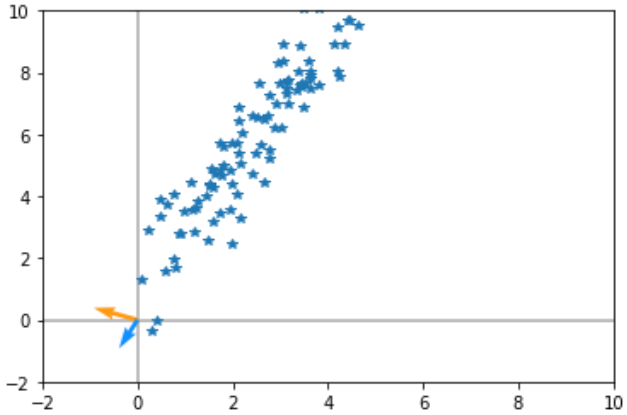
- The rotation transformed our dataset that have now the more variance on one of the basis axis.

What if we don't center the data????

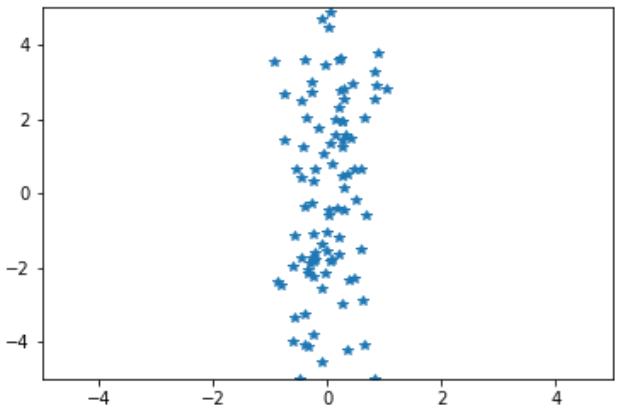
In [336]:
```
eigVals,  eigVecs = np.linalg.eig(X.T.dot(X))
print("Eigen Vectors = \n", eigVecs)
print("Eigen Values = \n",np.diag(eigVals))
```

```
Eigen Vectors =
 [[-0.9219176  -0.38738604]
 [ 0.38738604 -0.9219176 ]]
Eigen Values =
 [[  18.71468582    0.        ]
 [   0.         5085.69996889]]
```

In [337]:
```
orange = '#FF9A13'
blue = '#1190FF'
plotVectors(eigVecs.T, [orange, blue])
plt.plot(X[:,0], X[:,1], '*')
plt.xlim(-2, 10)
plt.ylim(-2, 10)
plt.show()
```



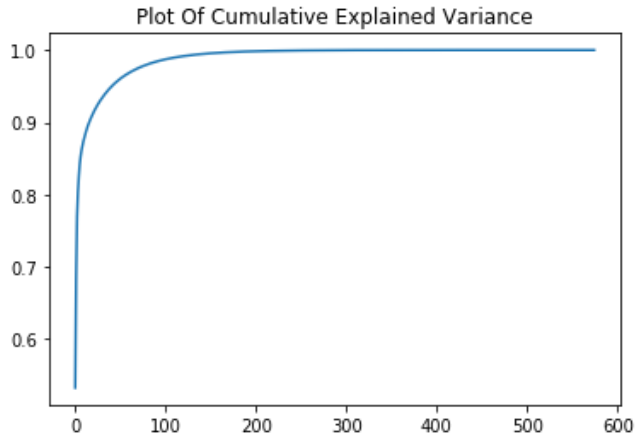
In [338]:
```
X_new = eigVecs.T.dot(X_center.T)
plt.plot(X_new[0, :], X_new[1, :], '*')
plt.xlim(-5, 5)
plt.ylim(-5, 5)
plt.show()
```



7.3) PCA using Sklearn

In [413]:
```
from sklearn.decomposition import PCA
pca = PCA(n_components=img.shape[1])
new_dim = pca.fit_transform(img)
```

In [418]:
```
#Choosing Number Of Components
plt.plot(pca.explained_variance_ratio_.cumsum())
plt.title("Plot Of Cumulative Explained Variance")
plt.show()
```



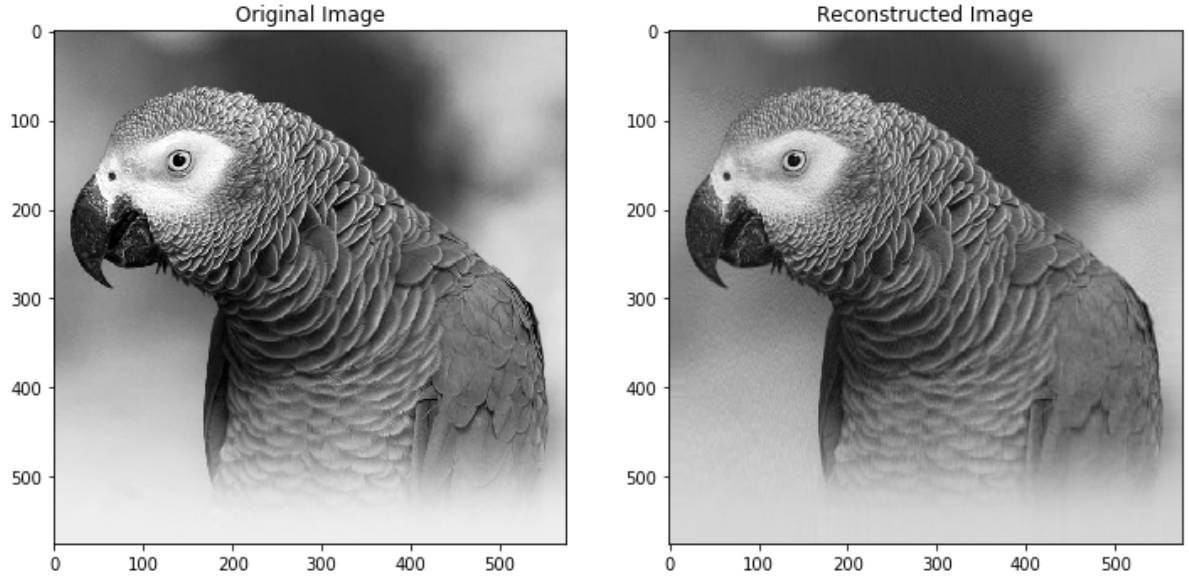
- We will keep around 100 components

In [422]:
```
pca = PCA(n_components=100)
new_dim = pca.fit_transform(img)
print("Shape of new data = ",new_dim.shape)
print("Reconstructing the Original Data : ")
fin_img = pca.inverse_transform(new_dim)
```

```
Shape of new data =  (575, 100)
Reconstructing the Original Data :
```

In [ ]:
```
f = plt.figure()
f.add_subplot(1,2, 1)
plt.imshow(np.rot90(imgLr,2))
f.add_subplot(1,2, 2)
plt.imshow(np.rot90(imgRr,2))
plt.show(block=True)
```

In [434]:
```
f = plt.figure(figsize=(12, 12))
f.add_subplot(1,2, 1)
plt.imshow(img, cmap='gray')
plt.title("Original Image")
f.add_subplot(1,2, 2)
plt.imshow(fin_img, cmap='gray')
plt.title("Reconstructed Image")
plt.show(block=True)
```



7.5) Relationship Between SVD and PCA:

- PCA and SVD are intuitively related.
- In PCA, we compute Eigen Values and Eigen vectors of the Covariance Matrix ie the product of XX^T Where X is the Data Matrix.
- Covariance Matrix is Symmetric and also diagonalizable and we can normalize eigen vectors such that they are orthonormal.

$$XX^T = WDW^T$$

- Now applying SVD to X, we get:

$$X = UDV^T$$

- Create Covariance Matrix from this decomposition gives :

$$XX^T = (UDV^T)(UDV^T)^T$$

$$= (UDV^T)(VDU^T)$$

- Since **V** is Orthonormal $V^T.V = 1$

$$XX^T = UD^2V^T$$

- The square root of Eighen Values of XX^T are singular values of **X**
- We can use SVD to perform SVD to perform PCA.
- This is better than forming the Covariance Matrix since formation of XX^T causes loss of precision.
- Additional Reference : https://arxiv.org/pdf/1404.1100.pdf (https://arxiv.org/pdf/1404.1100.pdf)