Numerical Examples: Basis for Grayscale Images (Python)

Help Center (https://accounts.coursera.org/i/zendesk/courserahelp?

return_to=https://courserahelp.zendesk.com/hc/)

The goal of this example is to illustrate the theoretical concepts studied in class (vector space, scalar product, basis and approximation) with a specific class of signals, grayscale images. Consider the following grayscale image of size 64 by 64 pixels.

In [1]:
```
from IPython.display import Image
Image(filename='Num_Ex_3/camera_blurred_big.jpg')
```

Out[1]:

It is represented by a square 64×64 matrix I, where each element corresponds to the intensity of a pixel. The matrix `I` is determined from the image using the `imread` function.

In [2]:
```
%pylab inline
import matplotlib.pylab as plt
I = np.array(plt.imread('Num_Ex_3/camera_blurred.jpg'), dtype=float64)
```

```
Populating the interactive namespace from numpy and matplotlib
```

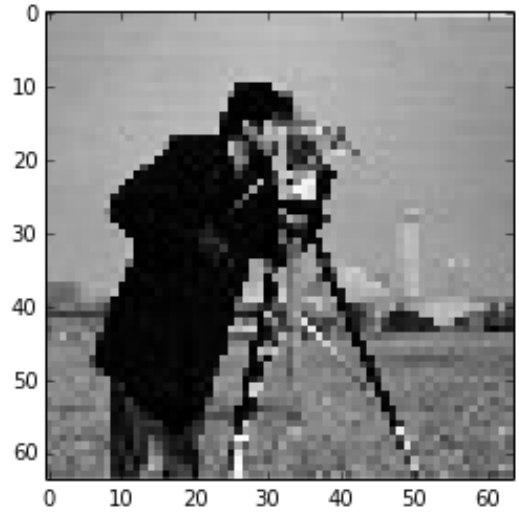
For example, the first column of this image is a 64×1 vector

In [3]:
```
I[:,0]
```

Out[3]:
```
array([ 156.,  157.,  157.,  152.,  154.,  155.,  151.,  157.,  152.,
        155.,  158.,  159.,  159.,  160.,  160.,  161.,  155.,  160.,
        161.,  161.,  164.,  162.,  160.,  162.,  158.,  160.,  158.,
        157.,  160.,  160.,  159.,  158.,  163.,  162.,  162.,  157.,
        160.,  114.,  114.,  103.,   88.,   62.,  109.,   82.,  108.,
        128.,  138.,  140.,  136.,  128.,  122.,  137.,  147.,  114.,
        114.,  144.,  112.,  115.,  117.,  131.,  112.,  141.,   99.,
        97.])
```

Conversely, one can display the array as an image by using the `pylab.imshow` function from matplotlib

In [4]:
```
import matplotlib.pylab as plt
plt.imshow(I, cmap=plt.cm.gray, interpolation='none')  #The cmap=plt.cm.
gray renders the image in gray
plt.show()
```



We can define two operations on this class of signals, the addition and multiplication by a scalar. The addition of two images is defined like the standard matrix addition

$$I1 + I2 = \begin{pmatrix} I1_{1,1} & \dots & I1_{1,64} \\ \vdots & & \vdots \\ I1_{64,1} & \dots & I1_{64,64} \end{pmatrix} + \begin{pmatrix} I2_{1,1} & \dots & I2_{1,64} \\ \vdots & & \vdots \\ I2_{64,1} & \dots & I2_{64,64} \end{pmatrix} = \begin{pmatrix} I1_{1,1} + I2_{1,1} & \dots & I1_{1,64} + I2_{1,64} \\ \vdots & & \vdots \\ I1_{64,1} + I2_{64,1} & \dots & I1_{64,64} + I2_{64,64} \end{pmatrix}.$$

The multiplication by a scalar is also defined like the corresponding matrix operation

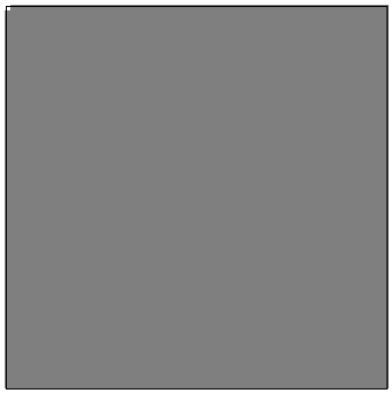
$$\alpha I1 = \alpha \begin{pmatrix} I1_{1,1} & \dots & I1_{1,64} \\ \vdots & & \vdots \\ I1_{64,1} & \dots & I1_{64,64} \end{pmatrix} = \begin{pmatrix} \alpha I1_{1,1} & \dots & \alpha I1_{1,64} \\ \vdots & & \vdots \\ \alpha I1_{64,1} & \dots & \alpha I1_{64,64} \end{pmatrix}.$$

We can verify that the space of 64 by 64 images endowed with these two operations defines an appropriate vector space, that is that it satisfied properties (1) to (8) in slide 42. We can also endow this vector space with a scalar product defined as

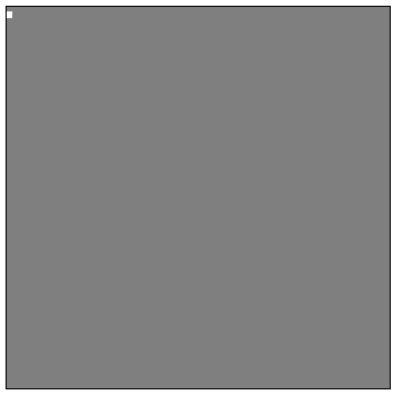
$$\langle I1, I2 \rangle = \sum_{n=1}^{64} \sum_{m=1}^{64} I1_{n,m} I2_{n,m}.$$

Observe that this definition of the scalar product corresponds to the ubiquitous one in $\mathbb{R}^{4096}$. This result is obtained by stacking the columns of the 64×64 matrix in a $64 * 64 = 4096 \times 1$ vector.

A natural (canonical) basis for this space is formed by the set of matrices where only one element equals to one and all the others equal 0. The following table represents these basis vectors. Of course, it is cumbersome to enumerate them all. We display the two first ones, e_1 and e_2, as well as the last one e_{4096}. This gives already a good idea. On each row, we represent, on the right, the basis vector and, on the left, the corresponding image. In these images, a black pixel corresponds to a value of -1, a gray one to 0 and a white one to 1.

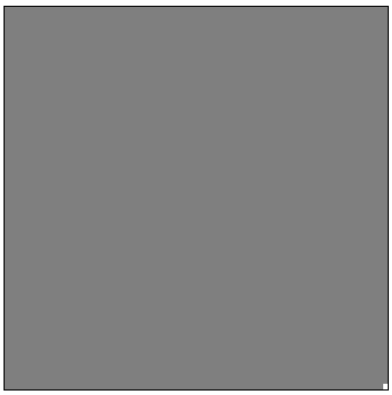


$$e_1 = \begin{pmatrix} 1 & \cdots & 0 \\ 0 & \cdots & 0 \\ \vdots & & \vdots \\ 0 & \cdots & 0 \end{pmatrix}$$



$$e_2 = \begin{pmatrix} 0 & \cdots & 0 \\ 1 & \cdots & 0 \\ \vdots & & \vdots \\ 0 & \cdots & 0 \end{pmatrix}$$

and so on until



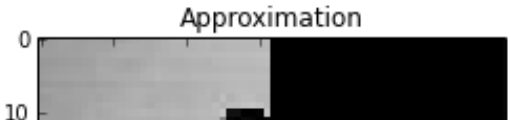
$$e_{4096} = \begin{pmatrix} 0 & \cdots & 0 \\ 0 & \cdots & 0 \\ \vdots & & \vdots \\ 0 & \cdots & 1 \end{pmatrix}$$

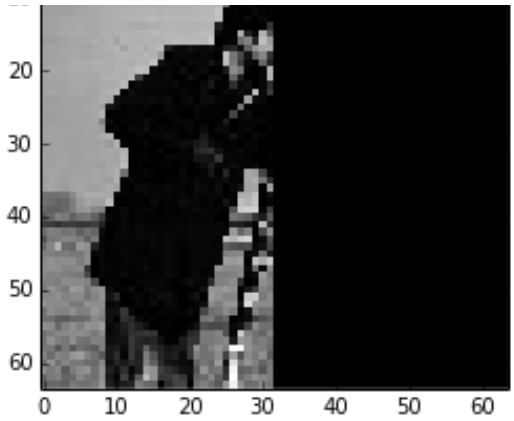
It looks as if all images are identical. Look more closely, you will notice that there is in each a small white pixel, on the 1st row and 1st column in the first image, for example.

Suppose we would like to transmit this image over a communication channel. At the sender side, the image is projected in the canonical basis and the resulting coefficients are sent. This simply correspond to sending individually the intensity of each pixel. Suppose there is an error during transmission such that only the first half of the coefficients is correctly transmitted. The received images is only an approximation of the original one and the missing pixels are replaced by a 0 correspond to holes in the image (in this case the entire right half)

We can compute the distance between the two images, i.e., the norm of the error.

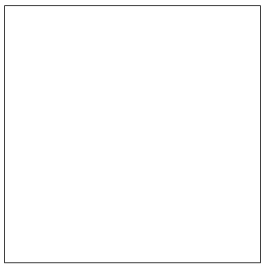
In [5]:
```
# Initialization of image
import matplotlib.pylab as plt
I_approx = np.array(plt.imread('Num_Ex_3/camera_blurred.jpg'), dtype=flo
at64)

I_approx[:,I_approx.shape[1]/2:] = 0

plt.imshow(I_approx, cmap=plt.cm.gray, interpolation='none')
plt.title('Approximation')
plt.show()

import math as m
# Error calculation
error = I - I_approx
distance = m.sqrt(sum(sum(error*error)))
print 'The distance between the original and approximate image is: ',dis
tance
```



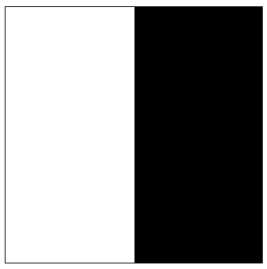



```
The distance between the original and approximate image is:  6586.036972
26
```

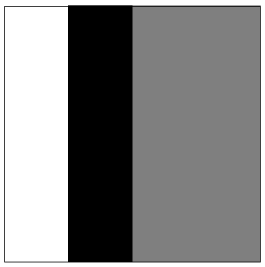
The question is whether we can do better by using another basis than the canonical one. The answer is yes. Consider for example the Haar basis defined by the set of matrices. The following table represents the first fours basis vectors $\psi_1, \ldots, \psi_4$. Again, we display on each row, on the right, the basis vector and, on the left, the corresponding image. In each image, a black pixel corresponds to a value of -1, a gray one to 0 and a white one to 1.



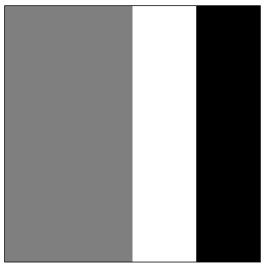
$$\psi_1 = \begin{pmatrix} 1 & \cdots & 1 \\ \vdots & & \vdots \\ 1 & \cdots & 1 \end{pmatrix}$$



$$\psi_2 = \begin{pmatrix} 1 & \ldots & 1 & -1 & \cdots & -1 \\ \vdots & & \vdots & \vdots & & \vdots \\ 1 & \cdots & 1 & -1 & \cdots & -1 \end{pmatrix}$$



$$\psi_3 = \begin{pmatrix} 1 & \ldots & 1 & -1 & \cdots & -1 & 0 & \cdots & \cdots & 0 \\ \vdots & & \vdots & \vdots & & \vdots & \vdots & & & \vdots \\ 1 & \cdots & 1 & -1 & \cdots & -1 & 0 & \cdots & \cdots & 0 \end{pmatrix}$$



$$\psi_4 = \begin{pmatrix} 0 & \cdots & \cdots & 0 & 1 & \ldots & 1 & -1 & \cdots & -1 \\ \vdots & & & \vdots & \vdots & & \vdots & \vdots & & \vdots \\ 0 & \cdots & \cdots & 0 & 1 & \cdots & 1 & -1 & \cdots & -1 \end{pmatrix}$$

We observe that Haar basis is composed of the scaled and shifted version of the same signal

$$\psi(t) = \begin{cases} 1 \text{ if } 0 \leq t < 1/2 \\ -1 \text{ if } 1/2 \leq t \leq 1 \\ 0 \text{ otherwise} \end{cases}$$

We can verify that this basis is indeed orthogonal by computing the scalar product between any two matrices ψ_i and ψ_j, for example,

```
(psi_i * psi_j).sum() = 0 // when orthogonal
```

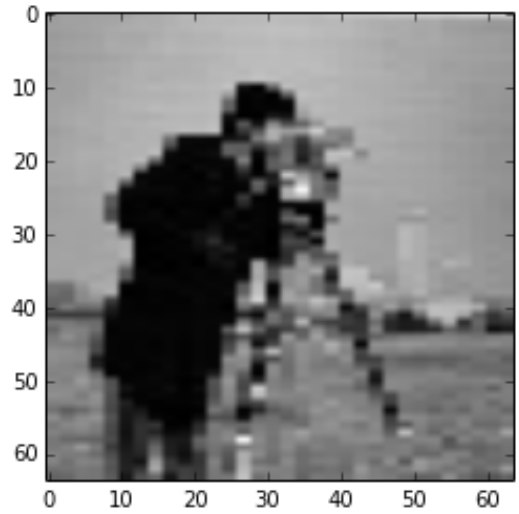
As before we project the image onto this basis and send the coefficients over a communication channel that looses the second half of the coefficients. This is implemented in the following code.

In [6]:
```
def haar(N):

    #Assuming N is a power of 2
    import numpy as np
    import math as m
    import scipy as sc
    h = np.zeros((N,N), dtype = float)
    h[0] = np.ones(N)/m.sqrt(N)
    for k in range(1,N) :

        p = sc.fix(m.log(k)/m.log(2))
        q = float(k - pow(2,p))
        k1 = float(pow(2,p))
        t1 = float(N / k1)
        k2 = float(pow(2,p+1))
        t2 = float(N / k2)

        for i in range(1,int(sc.fix(t2))+1):
            h[k,i+q*t1-1] = pow(2,(p/2))/m.sqrt(N)
            h[k,i+q*t1+t2-1] = -pow(2,(p/2))/m.sqrt(N)

    return h
```

```
import numpy as np

#Load image
import matplotlib.pylab as plt
I = np.array(plt.imread('Num_Ex_3/camera_blurred.jpg'), dtype='float64')
size = I.shape

#Arrange image in column vector
I = I.flatten()
#Generate Haar basis vector (rows of H)
H = haar(4096)

#Project image on the new basis
I_Haar = np.dot(H,I)

#Remove the second half of the coefficient
I_Haar[2048 : 4095] = 0

#Recover the image by inverting change of basis
I_Haar = np.dot(H.T,I_Haar)

#Rearrange pixels of the image
I_Haar = I_Haar.reshape(size)
```

In this case, the image recovered at the receiver side looks like

In [7]:
```
imshow(I_Haar, cmap=plt.cm.gray)
show()
```



with a distance to the original image of

In [8]:
```
I = np.array(plt.imread('Num_Ex_3/camera_blurred.jpg'), dtype='float64')
error_h = I - I_Haar
import math as m
distance = m.sqrt(sum(sum(error_h*error_h)))
print 'The distance between the original image and the Haar approximatio
n is ',distance
```

```
The distance between the original image and the Haar approximation is  1
319.4676957
```

This happens because the Haar basis decomposes the image into a successive approximation. The first coefficients in the basis correspond to a coarse and smoothed version of the image (low frequency) whereas the subsequent coefficients represent the details in the image (high frequency). Then, in our scenario, we recover a coarser yet complete version of the image.

Gram-Schmidt orthonormalization procedure

In the previous example, we have seen how an orthonormal basis is used to compute a successive approximation of a vector. But what happens if the set of vector spanning the space under consideration is not orthonormal? The Gram-Schmidt orthonormalization procedure enables to find the orthonormal base of this space.

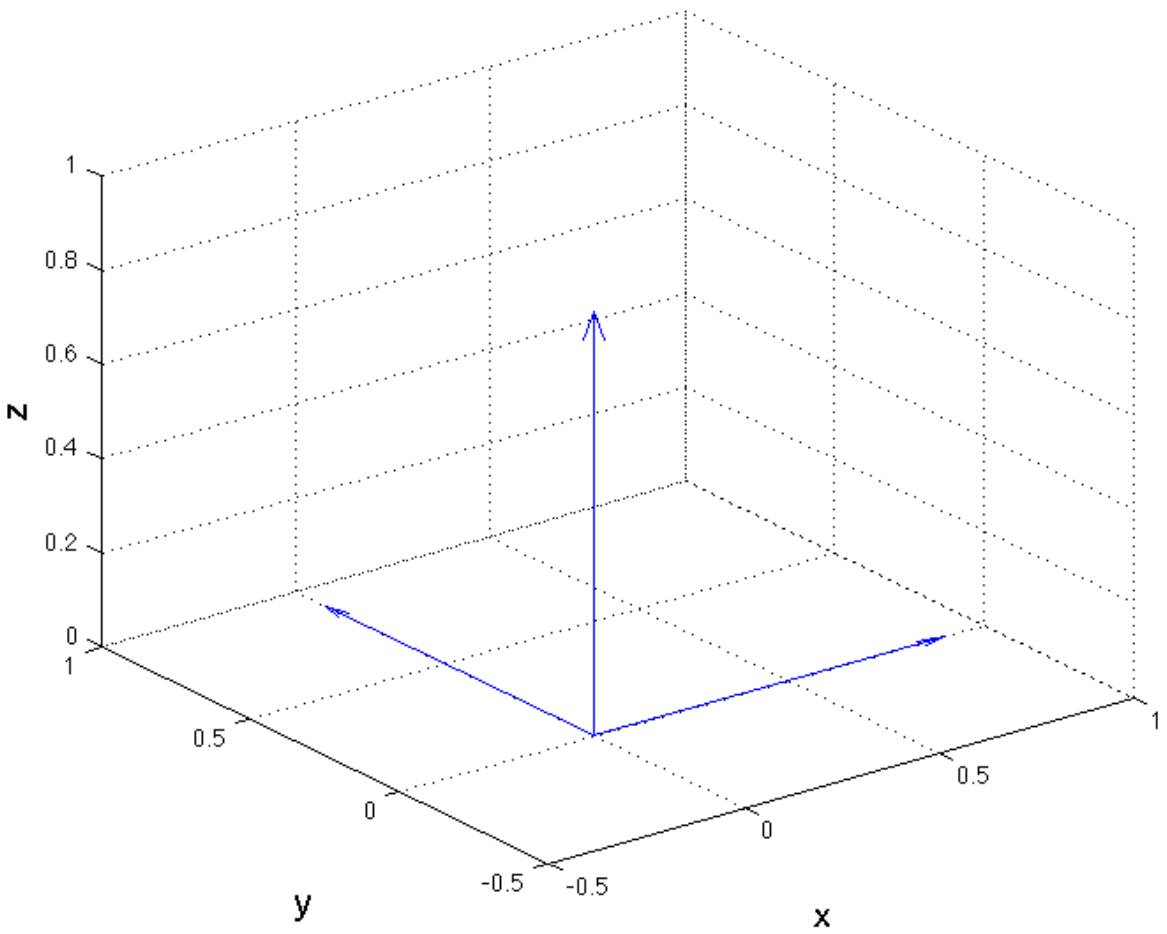
In [9]:
```
def gs_orthonormalization(V) :

    #V is a matrix where each column contains
    #the vectors spanning the space of which we want to compute the orth
onormal base
    #Will return a matrix where each column contains an ortho-normal vec
tor of the base of the space

    numberLines = V.shape[0]
    numberColumns = V.shape[1]

    #U is a matrix containing the orthogonal vectors (non normalized)
    from numpy.linalg import norm
    import numpy as np
    U = np.zeros((numberLines,numberColumns))
    R = np.zeros((numberLines,numberColumns))

    for indexColumn in range(0,numberColumns) :
```

```
        U[:,indexColumn] = V[:,indexColumn]

        for index in range(0,indexColumn):
            R[index,indexColumn] = np.dot(U[:,index],V[:,indexColumn])
            U[:,indexColumn] = U[:,indexColumn] - R[index,indexColumn]*U
[:,index]

        R[indexColumn,indexColumn] = norm(U[:,indexColumn])
        U[:,indexColumn] = U[:,indexColumn] / float(R[indexColumn, index
Column])

    return U
```

Let us consider a simple yet interesting example. Take the orthonormal base in $\mathbb{R}^3$, which reads

$$e_x = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, e_y = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, e_z = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}.$$

The figure below depicts the vectors of the orthogonal basis.



The following rotation matrices

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

$$R_y(\alpha) = \begin{bmatrix} \cos\alpha & 0 & \sin\alpha \\ 0 & 1 & 0 \\ -\sin\alpha & 0 & \cos\alpha \end{bmatrix}$$

$$R_z(\beta) = \begin{bmatrix} \cos\beta & -\sin\beta & 0 \\ \sin\beta & \cos\beta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

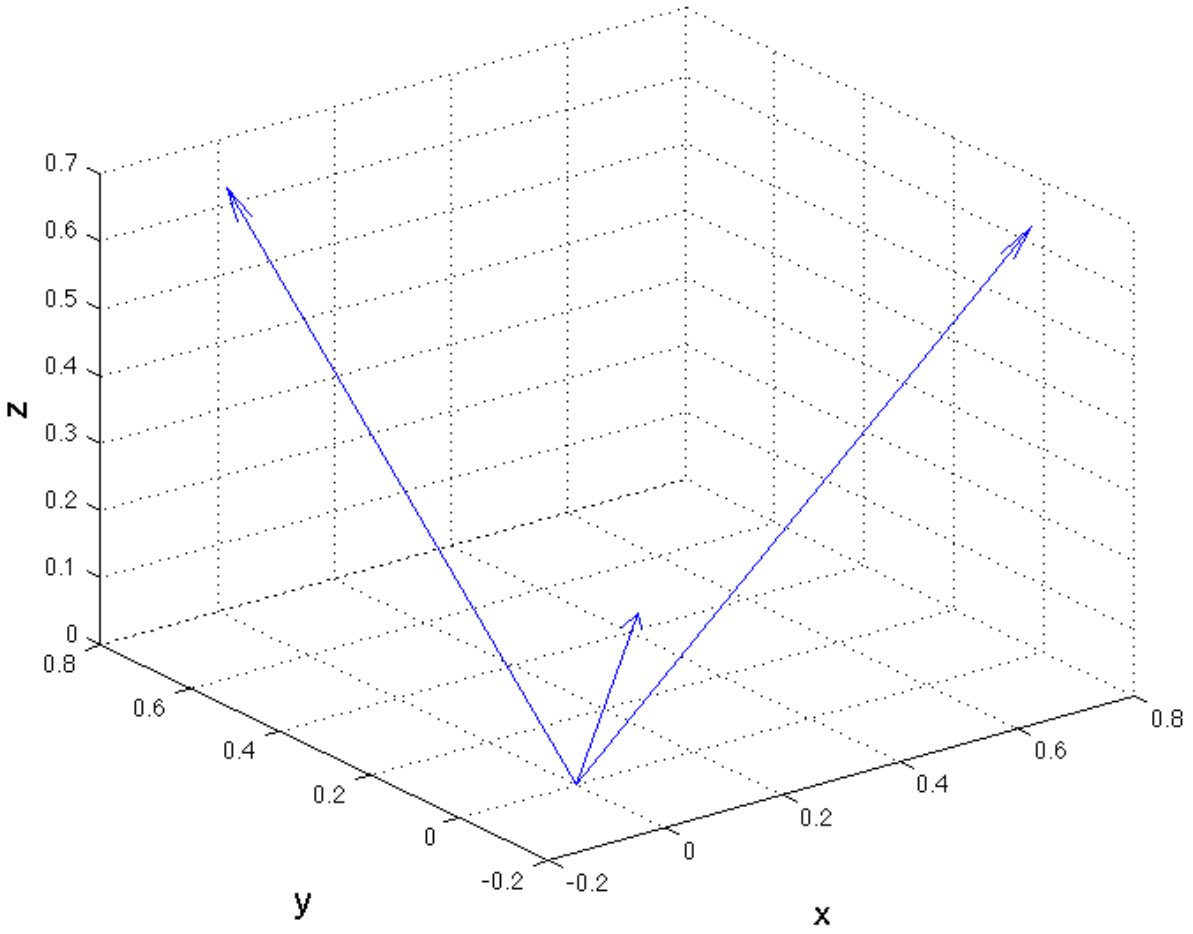
perform a counter-clock wise rotation of an angle θ, α, and β with respect to the axis x, y, and z, respectively.

Let's rotate the e_x w.r.t. the z axis, e_y w.r.t. the x axis, and e_z w.r.t. the y axis, with $\theta = \pi/3$, $\alpha = \pi/4$, and $\beta = \pi/6$

$$v_1 = R_z(\beta)e_x = \begin{bmatrix} 0.8660 \\ 0.5000 \\ 0 \end{bmatrix}$$

$$v_2 = R_x(\theta)e_y = \begin{bmatrix} 0 \\ 0.5000 \\ 0.8660 \end{bmatrix}$$

$$v_3 = R_y(\alpha)e_z = \begin{bmatrix} 0.7071 \\ 0 \\ 0.7071 \end{bmatrix}$$

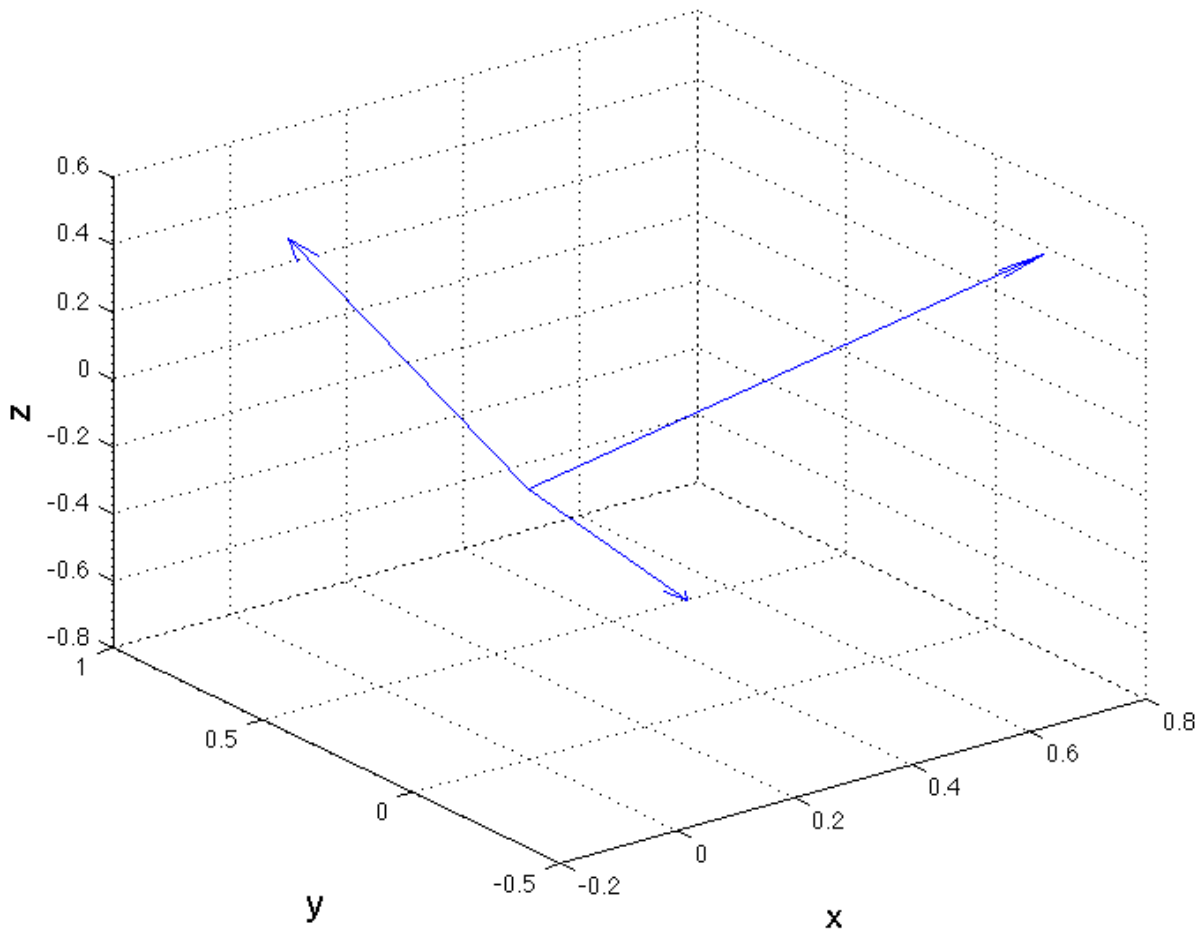


The so obtained three vectors, depicted in the figure above, do not form an orthogonal basis anymore. For instance $< v_1, v_2 >= 0.25$.

We can use the Gram-Schmidt procedure to re-obtain an orthonormal basis. By feeding the procedure with the matrix composed of the three vectors v_1, v_2, and v_3 we obtain

$$E = \begin{bmatrix} 0.866 & -0.2236 & 0.4472 \\ 0.5 & 0.3873 & -0.7746 \\ 0 & 0.8944 & 0.4472 \end{bmatrix}.$$

The figure below presents the new orthogonal vectors.



One can easily check that the colums of the matrix E form an orthonormal basis, i.e.

$$E^T E = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Finally, let us check the code:

In [10]:
```
v1 = np.array([[0.866, 0.5, 0.0]]).T
v2 = np.array([[0.0, 0.5, 0.866]]).T
```

```
v3 = np.array([[0.7071, 0.0, 0.7071]]).T
V = np.concatenate((v1, v2, v3), axis=1)
print 'matrix V'
print V

E = gs_orthonormalization(V)

print 'Orthonormalized basis:'
print E
print 'Check the orthonormality:',(np.dot(E.T,E) - np.eye(3)).sum()
```

```
matrix V
[[ 0.866   0.      0.7071]
 [ 0.5     0.5     0.    ]
 [ 0.      0.866   0.7071]]
Orthonormalized basis:
[[ 0.86601905 -0.22361565  0.44722147]
 [ 0.500011    0.38730231 -0.77458758]
 [ 0.          0.89442326  0.44722147]]
Check the orthonormality: 1.66533453694e-16
```

Exercises

Consider the three vectors

$$v_1 = \begin{bmatrix} 0.8660 \\ 0.5000 \\ 0 \end{bmatrix}$$

$$v_2 = \begin{bmatrix} 0 \\ 0.5000 \\ 0.8660 \end{bmatrix}$$

$$v_3 = \begin{bmatrix} 1.7320 \\ 3.0000 \\ 3.4640 \end{bmatrix}$$

Apply the Gram-Schmidt process on such vectors and provide the output matrix E.

Does the output matrix E represent a set of orthonormal vectors?

If yes, why?

1. dot(E.T, E) == I
2. det(E) == 0
3. rank(E) == 3

If not, why?

1. dot(E.T, E) is not the identity matrix
2. det(E) > 0
3. rank(E)> 1

Created Fri 11 Apr 20148:57 AM EDT

Last Modified Tue 20 Jan 2015 5:07 AM EST