Numerical Examples: Fourier Transforms (Python)

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

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

DFS & DFT

The discrete Fourier transform (DFT) of a signal is defined using the following formula:

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-j(2\pi/N)(n)(k)}, k = 0, \ldots, N-1.$$

Similar formula is used for the discrete Fourier series (DFS) of a periodic signal.

The DFS / DFT is a vector of complex numbers which can be represented via their magnitude $|X(n)|$ (absolute value) and phase $\arctan \frac{\mathrm{Im}(X(n))}{\mathrm{Re}(X(n))}$, $n = 0, \ldots, N-1$.

Transformation matrix

DFS / DFT can be easily implemented using the transform matrix W_N. We recall that for a signal of length N the transform matrix has as n-th row and k-th column element $W_N(n,k) = e^{-j(2\pi/N)(n)(k)} = W_N^{(n)(k)}$, with $n, k = 0, \ldots, N-1$, and there we denote $W_N = e^{-j(2\pi/N)}$.

In Python, a DFT matrix can be generated using the following code.

In [1]:
```
def dftmatrix(N):
    '''construct DFT matrix'''
    import numpy as np

    # create a 1xN matrix containing indices 0 to N-1
    a = np.expand_dims(np.arange(N), 0)

    # take advantage of numpy broadcasting to create the matrix
    WN = np.exp(-2j*np.pi*a.T*a/N)

    return WN
```

The transformation X of the signal x, i.e. $X(k)$; is then given by (here using a short constant signal), multiplying the (here) 3×3 DFT Matrix and the input signal (in its vector form).

In [2]:
```
%pylab inline

N = 6
WN = dftmatrix(N)

x = np.ones(N)
X = np.dot(WN, x)
```

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

The inverse transformation $x(n) = \frac{1}{N} \sum_{k=1}^{N} X(k) e^{j(2\pi/N)(n)(k)}$ is then computed using:

In [3]:
```
x2 = 1./N * np.dot(WN.T.conjugate(), X)
print 'Error: ',(np.abs(x - x2).sum())
print '(This small error is due to machine precision.)'
```

```
Error:  5.38784573873e-15
(This small error is due to machine precision.)
```

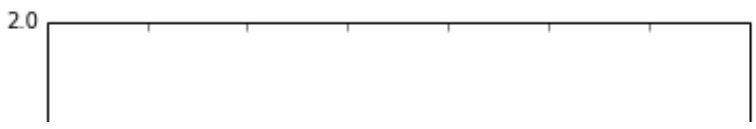
Magnitude and Phase

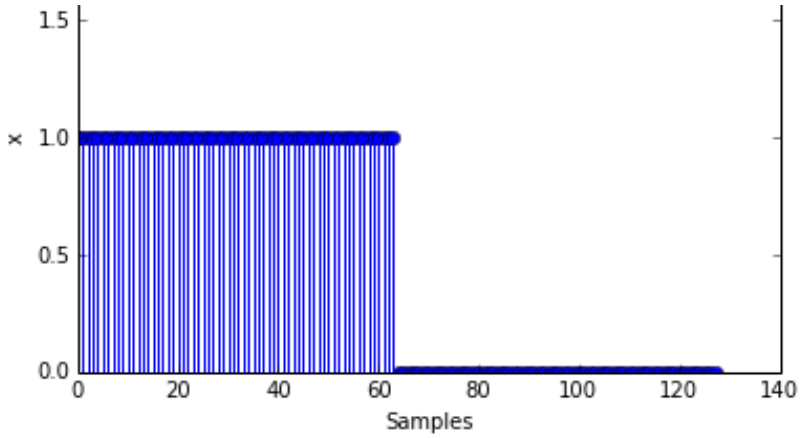
Let's start with a simple signal: unitary step function (periodic) of length N=128 (we shall see later what motivates us in choosing a power of 2).

In [4]:
```
x = np.append(np.ones(64),np.zeros(64))
```

We can see what this signal looks like in the next figure

In [5]:
```
y = np.linspace(0,127,128)
# The plotting is done by the pylab.stem function
stem(y,x)
xlabel('Samples')
ylabel('x')
ylim([0,2])
show()
```





The transformation matrix and the transform is then

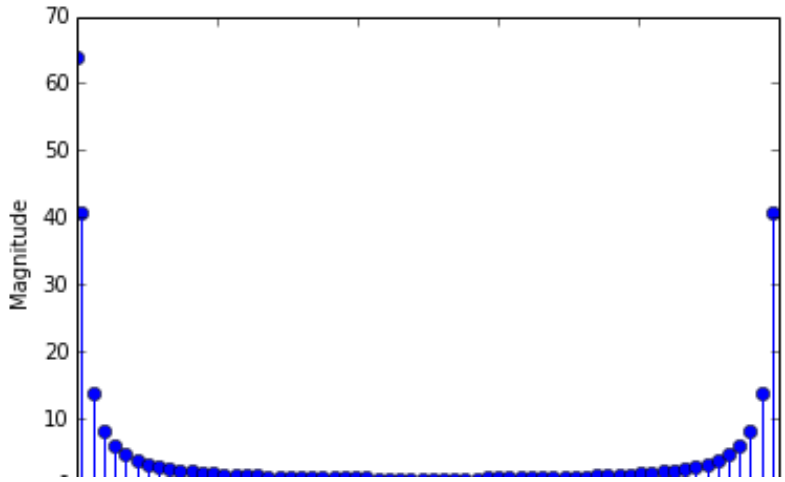
In [6]:
```
x = x.flatten();
N = 128
W128 = dftmatrix(N)
X  = np.dot(W128,x)
```

The stem plot of the magnitude (absolute value) of the DSF or DFT

In [7]:
```
magnitude = abs(X)
```

is given by

In [8]:
```
y = np.linspace(0,127,128)
y[:] = y[:]/len(y) #we are normalizing the frequencies
# Plotting is done by the pylab.stem function
stem(y,magnitude)
ylabel('Magnitude')
xlabel('Normalized Frequencies')
show()
```



When it comes to compute the phase we can use the Python command

```
In [9]: phase = np.angle(X)
```

Numerical intermezzo

It is important to notice that, when the phase is close to $k\pi$ with $k = 0, 1, 2, \ldots$, its computation is extremely sensitive to numerical errors.

Consider for instance the third coefficient $X(3)$ of the DFT / DFS of the step function.

If we theoretically compute such value, by remarking that is a geometric series $X(3) = \sum_{n=1}^{N/2} a^n$ with $a = e^{-j(2\pi/N)(n-1)(3-1)}$, we obtain $X(3) = \frac{1-e^{-j(2\pi/N)2N/2}}{1-e^{-j(2\pi/N)2}} = 0$, i.e., a null complex number to which we assign by definition a phase of 0 degrees. Such a result applies to all coefficients with odd index.

If we numerically compute such value

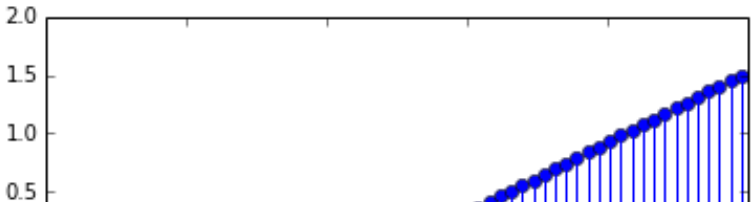
```
In [10]: X[2] = sum(W128[2])
         X[2]
```

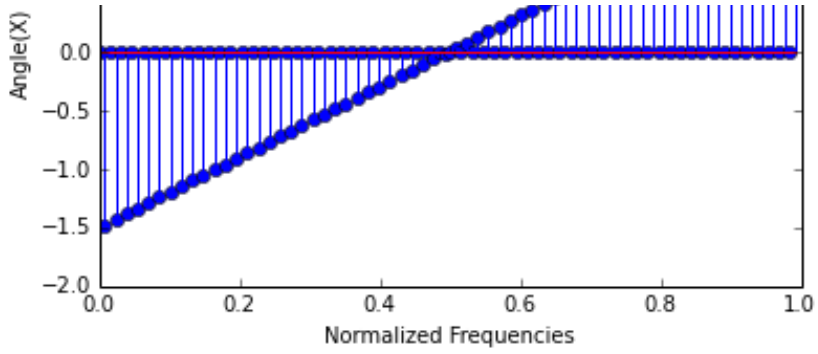
```
Out[10]: (-3.4416913763379853e-15+3.8441472227646045e-15j)
```

We obtain -3.44e-15 + 3.84e-15i which give us a phase of -0.7608 radian! Such a result is nothing but the numerical error of the sum computation.

Notice that the theoretical phase of X is

```
In [11]: Theoretical_phase = np.linspace(-1.5,1.5,128)
         Theoretical_phase[0:127:2] = 0
         stem(y,Theoretical_phase)
         ylabel('Angle(X)')
         xlabel('Normalized Frequencies')
         xlim([0,1])
         ylim([-2,2])
         show()
```



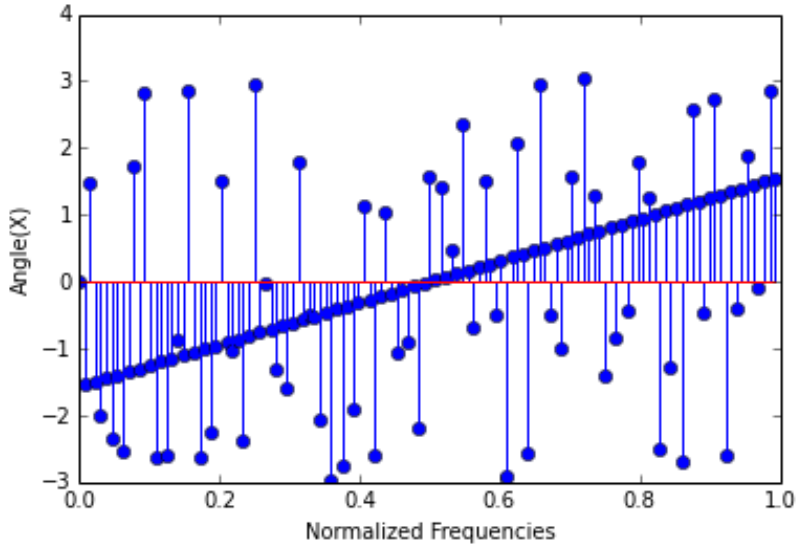


... a good paper and pencil exercise!

Back to our phase computation

So if we numerically compute the phase of a DFT / DFS obtained by the matrix multiplication and plot the phase we obtain

In [12]:
```
stem(y,phase)
ylabel('Angle(X)')
xlabel('Normalized Frequencies')
show()
```



where we can remark the phase computation error where the phase is supposed to be zero.

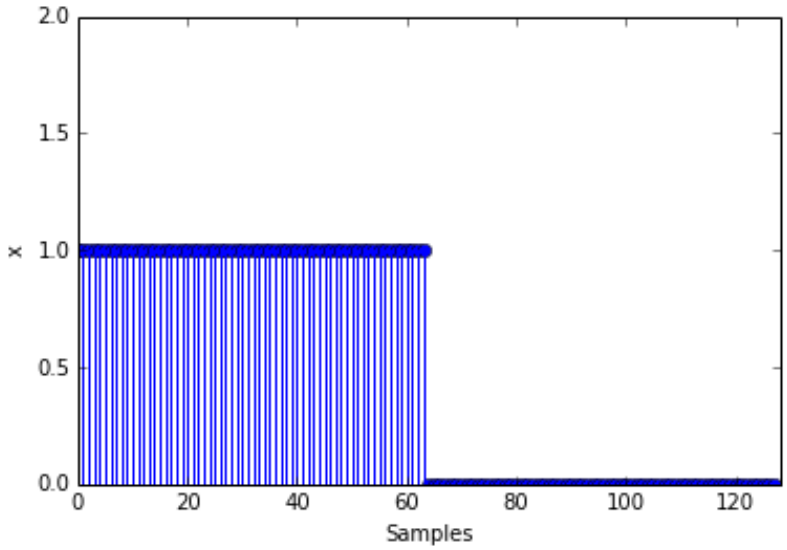
As we shall see in the following, there exists an optimized algorithm that besides speeding up the computation of the DFT / DFS; it avoids, under certain circumstances, numerical errors.

The inverse transformation, and it plot, read

In [13]:
```
xx = np.real(np.dot(np.conjugate(W128),X) *1/128)
```

Notice that, once again, we are here affected by numerical errors that add negligible imaginary parts to the results that is on the contrary supposed to be real. We shall therefore force the result to be real using the command np.real(), obtaining

In [14]:
```
stem(np.arange(xx.size),xx)
ylim([0,2])
xlim([0,128])
ylabel('x')
xlabel('Samples')
show()
```



The Role of the Fourier transform

Two birds with one stone

As discussed in the course, the Fourier transformation has a twofold reason to exist: (a) it enables to move into a domain where certain computations are easier to implement (e.g., convolution) and (b) it enables to represent the energy of the signal on the frequency domain rather than the time domain.The second reason to be can be put in evidence with a very simple numerical example.

Load the file `frequencyRepresentation.mat`. The signal x contained in the file is a sum of sinusoids, that is, is a sound composed of different tones. We would like to understand how many tones or sinusoids compose the signal.

To import a `.mat` file in Python, use the following commands

In [15]:
```
import scipy.io
```

```
mat = scipy.io.loadmat('frequencyRepresentation.mat')
print mat
```

```
{'x': array([[  7.30443561e-01,   1.32608160e+00,   1.68257383e+00, ...,
         -1.32608160e+00,  -7.30443561e-01,  -2.93915232e-14]]), '__vers
ion__': '1.0', '__header__': 'MATLAB 5.0 MAT-file, Platform: MACI64, Cre
ated on: Wed Dec 19 14:17:23 2012', '__globals__': []}
```

And then,

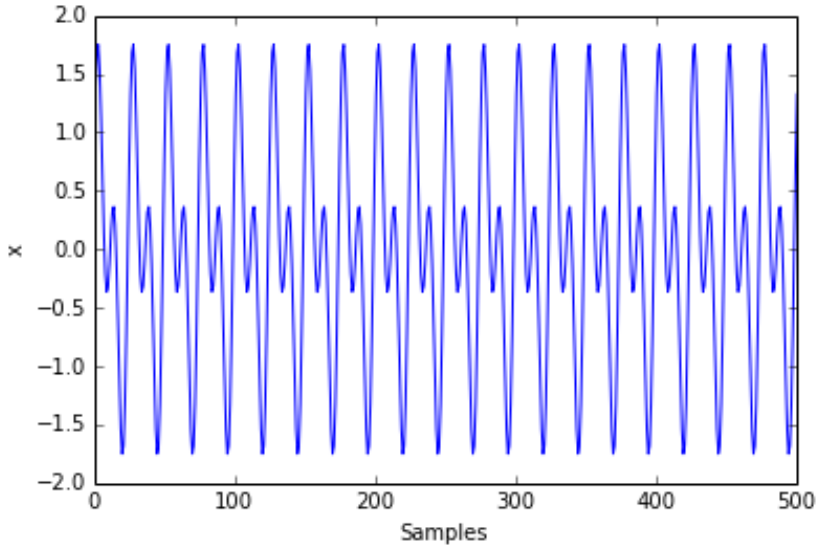
In [16]:
```
signal = mat['x']
print signal
type(signal)
```

```
[[  7.30443561e-01   1.32608160e+00   1.68257383e+00 ...,  -1.32608160e+
00
   -7.30443561e-01  -2.93915232e-14]]
```

Out[16]:
```
numpy.ndarray
```

In the time domain we can either visualize the signal using a basic plot function (the figure below only displays 500 samples out of the 4000),

In [17]:
```
signal = signal.reshape(signal.size,)
plot(np.arange(signal.size),signal)
xlim([0,500])
xlabel('Samples')
ylabel('x')
show()
```



or play it!

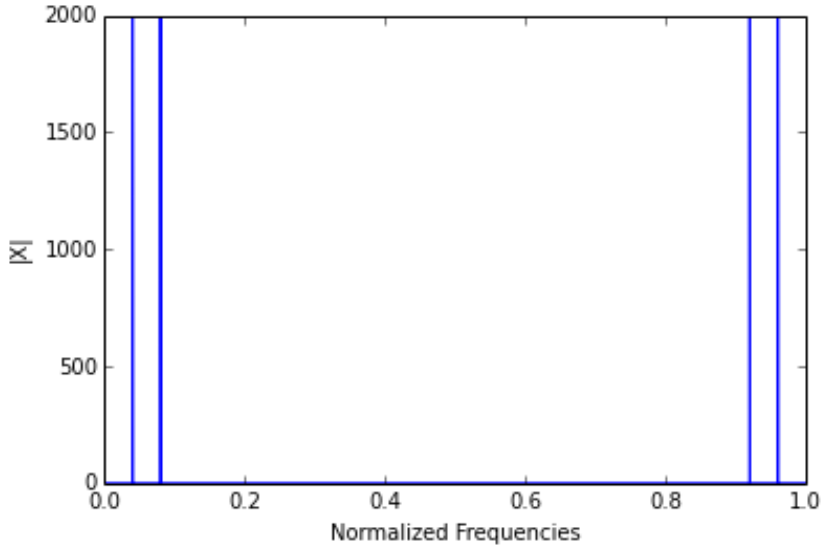
Actually, as you can remark, none of these approaches allow us to clearly understand how many sinusoids compose the signal.

Let's now take a look at the DFS /DFT. Once created, the transformation matrix (here $N = 4000$) `W4000`, we get

In [18]:
```
signal = signal.reshape(signal.size,1)
W4000 = dftmatrix(4000)
X = np.dot(W4000,signal)
```

The stem plot in normalized frequencies yields:

In [19]:
```
normFrequ = np.arange(1,X.size+1,dtype=float)/float(X.size)
plot(normFrequ,abs(X))
ylabel('|X|')
xlabel('Normalized Frequencies')
show()
```



DFS and DFT as projections on a base of complex sinusoids

In short, using the DFS or the DFT we are projecting a signal of length N on a basis of N complex sinusoids $W_N^{n,k}$ with $k = 1, \ldots, N$.

Let's see what happens if we reduce the dimension of the base, that is the number of sinusoids used to represent the signal.

One complex sinusoid (of order 0, i.e. a constant), taken from the first 128 samples of the `frequencyRepresentation` signal yields:

In [20]:
```
x = np.append(np.ones(64),np.zeros(64))
W128 = dftmatrix(128)
X1 = np.dot(W128[0],x)
xx1 = np.real(np.conjugate(W128[:,0])*X1*1/128)
```

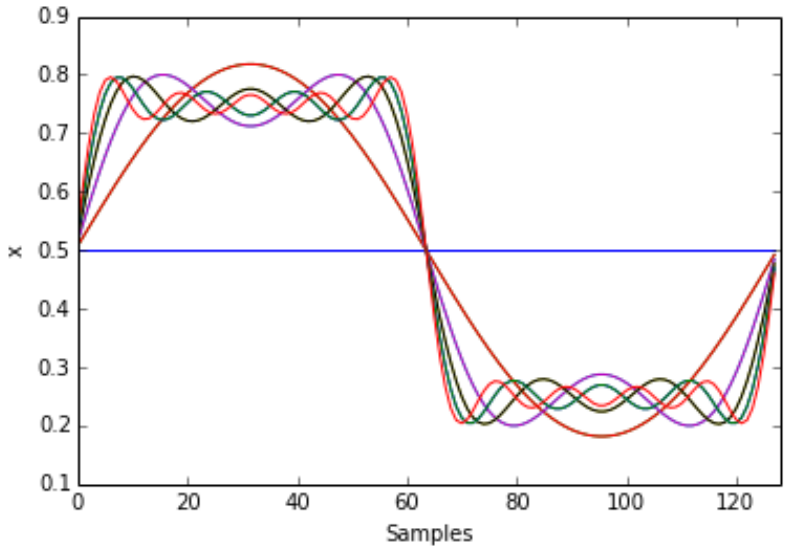
Two,three,... complex sinusoids will be:

In [21]:
```
X2 = np.dot(W128[0:2],x)
xx2 = np.real(np.dot(np.conjugate(W128[:,0:2]),X2)*1/128)

X3 = np.dot(W128[0:3],x)
xx3 = np.real(np.dot(np.conjugate(W128[:,0:3]),X3)*1/128)

X4 = np.dot(W128[0:4],x)
xx4 = np.real(np.dot(np.conjugate(W128[:,0:4]),X4)*1/128)

X5 = np.dot(W128[0:5],x)
xx5 = np.real(np.dot(np.conjugate(W128[:,0:5]),X5)*1/128)

X6 = np.dot(W128[0:6],x)
xx6 = np.real(np.dot(np.conjugate(W128[:,0:6]),X6)*1/128)

X7 = np.dot(W128[0:7],x)
xx7 = np.real(np.dot(np.conjugate(W128[:,0:7]),X7)*1/128)

X8 = np.dot(W128[0:8],x)
xx8 = np.real(np.dot(np.conjugate(W128[:,0:8]),X8)*1/128)

X9 = np.dot(W128[0:9],x)
xx9 = np.real(np.dot(np.conjugate(W128[:,0:9]),X9)*1/128)

X10 = np.dot(W128[0:10],x)
xx10 = np.real(np.dot(np.conjugate(W128[:,0:10]),X10)*1/128)
```

and so on, up to considering 10 complex sinusoids. If now we plot the signal representation using 1,2 and up to 10 sinusoids we obtain the following figure:

In [22]:
```
plot(np.arange(xx1.size),xx1)
plot(np.arange(xx2.size),xx2)
plot(np.arange(xx3.size),xx3)
```

```
plot(np.arange(xx4.size),xx4)
plot(np.arange(xx5.size),xx5)
plot(np.arange(xx6.size),xx6)
plot(np.arange(xx7.size),xx7)
plot(np.arange(xx8.size),xx8)
plot(np.arange(xx9.size),xx9)
plot(np.arange(xx10.size),xx10)
xlabel('Samples')
ylabel('x')
xlim([0,128])
show()
```



Three tones

Let us consider another example. We create a simple signal x composed of 3 tones x_1 , x_2 , and x_3

In [23]:
```
from scipy import constants as c
y = np.linspace(0,999,1000)
x1 = np.sin((y*2*c.pi*40)/1000)
x2 = np.sin((y*2*c.pi*80)/1000)
x3 = np.sin((y*2*c.pi*160)/1000)
```

We can put together these signals to get a final signal composed of $N = 5000$ samples, with silence in between the 3 different sine waves.

This signal can be played by using Audacity, or the winsoud module.

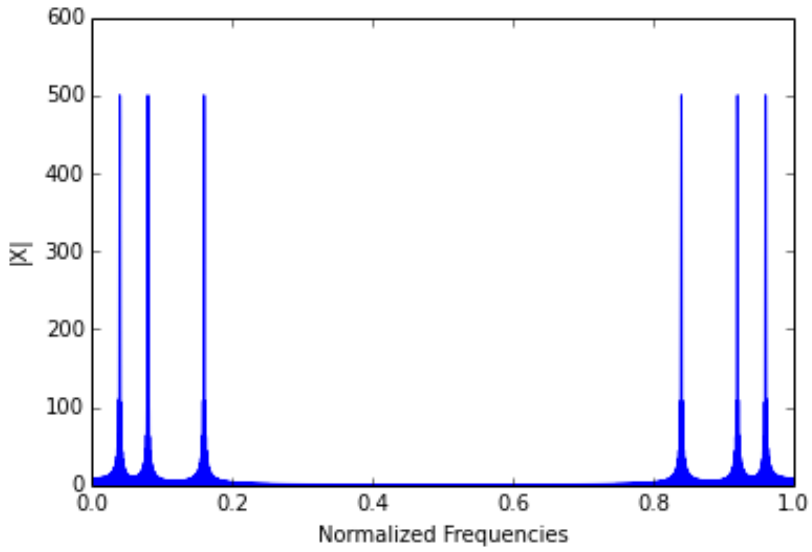
In [24]:
```
x = np.append(x1,[np.zeros(1000),x2,np.zeros(1000),x3])
```

Let's generate the Fourier matrix W_N (with $N = 5000$) and compute the DFS or DFT of the signal

In [25]:
```
W5000 = dftmatrix(5000)
X = np.dot(W5000,x)
```

The magnitude of the Fourier transformation is

In [26]:
```
normFrequ = np.arange(1,X.size+1,dtype=float)/float(X.size)
plot(normFrequ,abs(X))
ylabel('|X|')
xlabel('Normalized Frequencies')
show()
```



where we can clearly see the three tones.

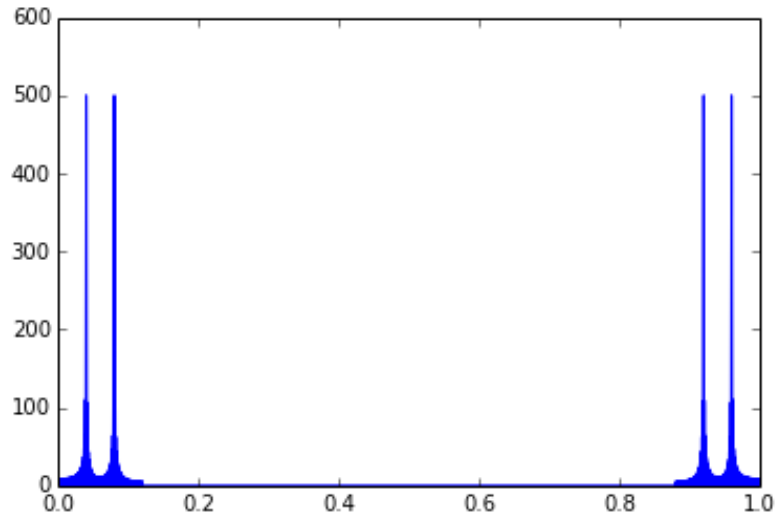
Let's now reduce the base over which the signal is represented, and take:

In [27]:
```
W5000reduced = np.append(W5000[0:600],W5000[4399:5000],axis=0)
iW5000reduced = np.conj(np.append(W5000[:,0:600],W5000[:,4399:5000],axis
=-1))

Xapprox = np.dot(W5000reduced,x)  # DFT/DFS on a reduced base
xapprox = np.dot(iW5000reduced,Xapprox)*1/5000 # inverse DFT/DFS of the
DFT/DFS on a reduced base
xapprox = np.real(xapprox) # real value of the result to get rid of imag
inary numerical errors
```

So `xapprox` is the signal x projected on a reduced base

Play the signal `xapprox` and take a look at its DFS / DFT:

In [28]:
```
X = np.dot(W5000,xapprox)

plot(np.arange(1,5001,dtype=float)/float(5000),abs(X))
show()
```



You will notice the difference!

Numerical optimization: Fast Fourier Transform - FFT

If we take a look at the matrix form of the DFS or DFT

```
X = dot(WN, x) # We assume here WN to have dimensions N times N and x to h
ave dimensions N times 1
```

we see that it requires $O(N^2)$ multiplications. As `N` increases, the computational burden is way too high. Already in the case of the previous example with $N = 5000$ the number of multiplication is not negligible.

The Fast Fourier Transform (FFT) is an algorithm that optimizes the computation of the DFS/DFT by reducing the number of multiplications to $2N \log_2 N$. Standard FFT algorithm requires N to be a power of 2, and, if it is not the case, the algorithm adds zeros (zero padding) to the signal to reach a number of samples that is a power of 2.

Python implements the Fast Fourier Transform using the following command:

In [29]:
```
from scipy import fftpack as f
X = f.fft(x)
```

The `fft()` function is also optimized with respect to numerical errors.

In the first numerical example presented in Fourier Transform, we have indeed chosen a step function of length 128 so as to fall into the framework of an optimized algorithm w.r.t. the computational burden and the numerical errors.

The FFT function also accepts an additional parameter `M` enabling to choose how many samples of `x` are used to compute the FFT

```
X = scipy.fftpack.fft(x,M)
```

Getting familiar with the FFT: Truncating and Zero Padding

Consider a signal x composed of N samples and let's use the function

```
X = fft(x,M)
```

to compute the FFT.

Truncating

If $M < N$ then the FFT is computed using the first M samples of x.

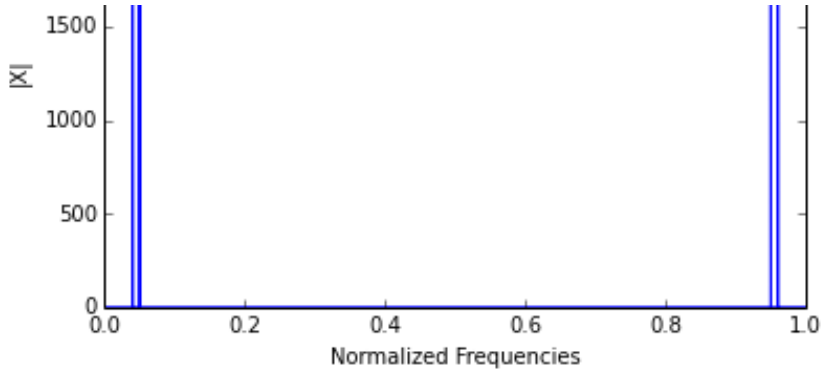
Let's take another two tones signal, with N=4000,

In [30]:
```
y = np.linspace(0,3999,4000)
x1 = np.sin((y*2*c.pi*40)/1000)
x2 = np.sin((y*2*c.pi*50)/1000)
x = x1 + x2
```

that we can of course play using the `scipy.io.wavfile.write` function.

Let's plot the spectrum and represent the frequency axis relatively to the number of samples, i.e., between 0 and 1. We are going to compute the fft using all the samples, that is

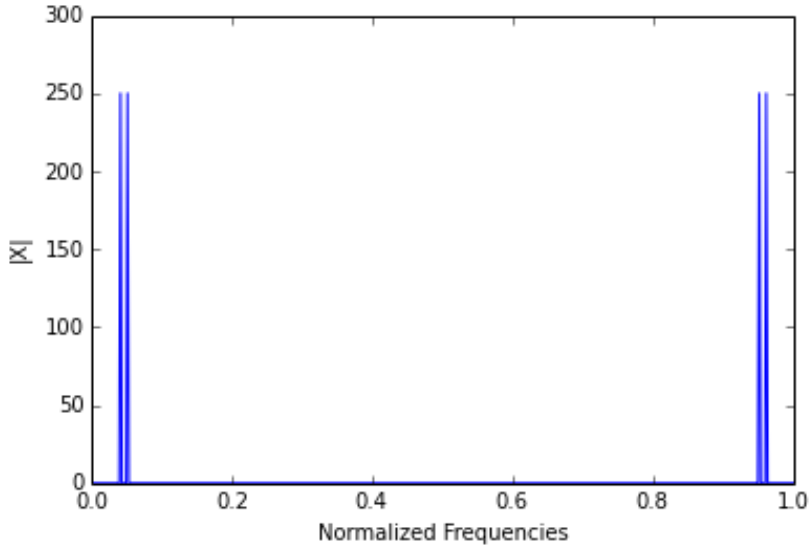
In [31]:
```
M = 4000
X = f.fft(x,M)
#Plot versus the normalized frequencies
normFrequ = np.arange(1,M+1,dtype=float)/float(M)
plot(normFrequ,abs(X))
ylabel('|X|')
xlabel('Normalized Frequencies')
show()
```



As expected, the spectrum shows the energy of the signal concentrated on two precise frequencies.

Let us now reduce the number of samples used to compute the DFS / DFT, and take for instance $M = 500$:

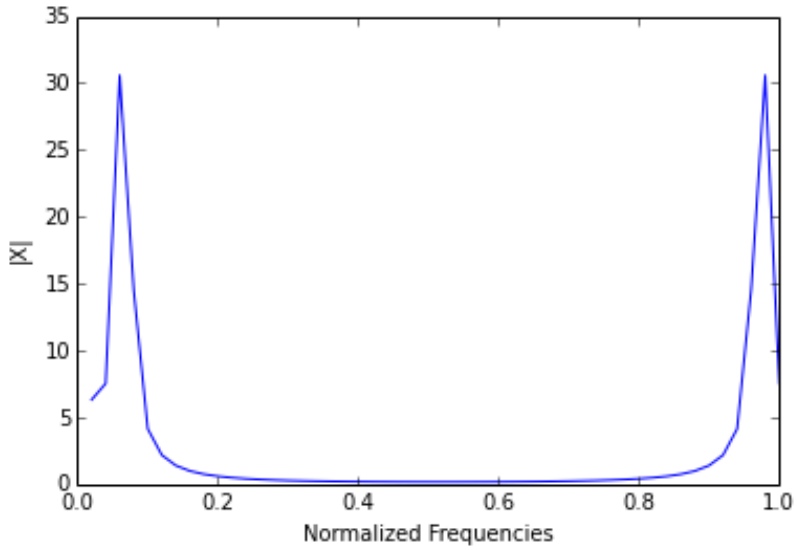
In [32]:
```
M = 500
X = f.fft(x,500)

#Plot versus the normalized frequencies
normFrequ = np.arange(1,M+1,dtype=float)/float(M)
plot(normFrequ,abs(X))
ylabel('|X|')
xlabel('Normalized Frequencies')
ylim([0,300])
show()
```



and $M = 50$

In [33]:
```
M = 50
```

```
X = f.fft(x,50)

#Plot versus the normalized frequencies
normFrequ = np.arange(1,M+1,dtype=float)/float(M)
plot(normFrequ,abs(X))
ylabel('|X|')
xlabel('Normalized Frequencies')
ylim([0,35])
show()
```



We can see that, as the number of samples decreases, it becomes more difficult to distignuish the two spectral lines that are on the contrary clearly visible on the DSF / DFT computed using all the 4000 samples. So the number of samples used to compute the DFS / DFT affects the resolution of the latter.

Zero Padding

As mentioned earlier, if $M < N$ the number of samples of the signal is increased to M by adding zeros.

Since truncating decreases the resolution of the DFS / DFT, one might think that zero padding increases such a resolution. But we shall see that this is not the case.

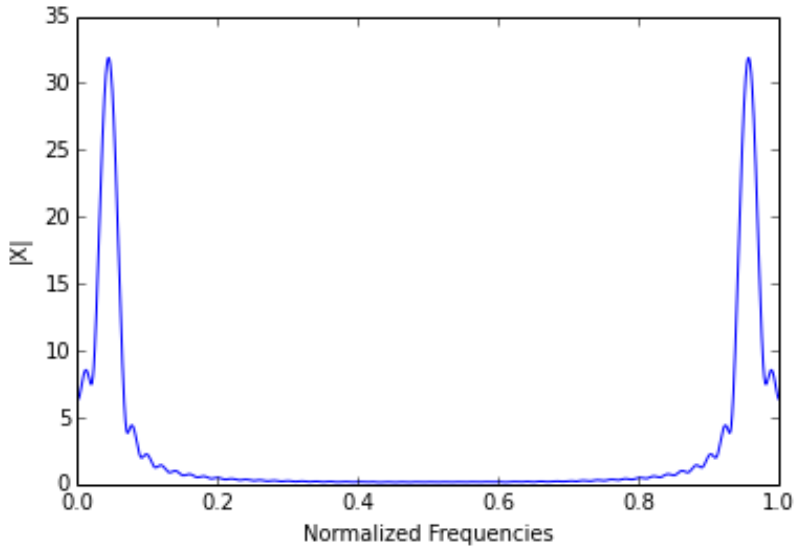
So let's take the DFS / DFT computed using only 50 samples and reconstruct the signal using such a DFS / DFT.

In [34]:
```
M = 50
X = f.fft(x,M)   # X is of length 50
x50 = f.ifft(X)
```

where we have used here the inverse FFT function `scipy.fftpack.ifft()`.

We have obtained a signal x_{50} of $N = 50$ samples and we compute the FFT with $M = 4000$, that is, we zero pad the signal so as to have the initial number of samples before computing the FFT.

In [35]:
```
M = 4000
X = f.fft(x50,M)

#Plot versus the normalized Frequencies
normFrequ = np.arange(1,M+1,dtype=float)/float(M)
plot(normFrequ,abs(X))
ylabel('|X|')
xlabel('Normalized Frequencies')
show()
```



As shown in the plot, the two spectral lines are still not distinguishable even if the FFT has now been computed using 4000 samples.

Created Fri 11 Apr 20148:57 AM EDT

Last Modified Tue 20 Jan 2015 5:08 AM EST