Lab 2 - The Discrete Fourier Transform

Instructor: Prof. Lillian Jane Ratliff

Teaching Assistants: Ashwin Srinivas Badrinath and Kevin Lin

Team Members:

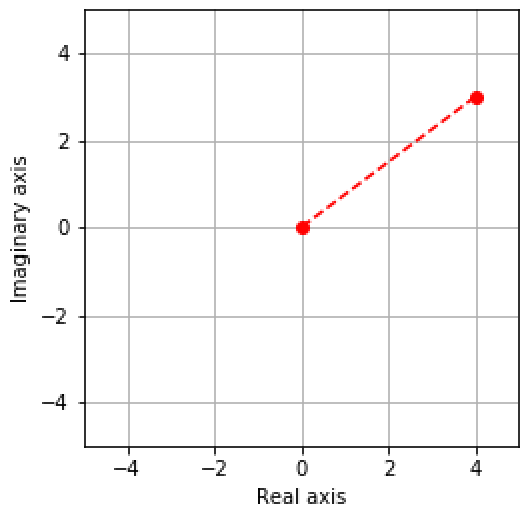
In [78]:
```
import numpy as np
import math
from math import *
import matplotlib
import matplotlib.pyplot as plt
from scipy.fftpack import fft, fftshift, ifft
```

1) Complex Numbers and Complex Sinusoids

In [61]:
```
# writing the complex number as real + imaginary
z1 = np.complex(4,3)
# using the function complex
z2 = np.complex(5,7)
z3 = np.complex(7,5)
print(z1)
print(z2)
print(z3)

# add the real part of 4+3j and the imaginary part of
# 5+7j and display the result
print(np.real(z1)+np.real(z2))
print(np.imag(z1)+np.imag(z2))

# subtract the imaginary part of 4+3j from the real part of
# 5+7j and display the result
print(np.imag(z1)-np.real(z2))
# multiply 4+3j and 4-3j and dislay the result
print((z1)*np.conj(z2))
# divide 7+5j and 7-5j and display the result
print(z3/np.conj(z3))
```

```
(4+3j)
(5+7j)
(7+5j)
9.0
10.0
-2.0
(41-13j)
(0.32432432432432434+0.945945945945946j)
```

In [62]:
```
# define a complex number
z = np.complex(4,3)

# obtain the real and imaginary parts of the complex number
real = np.real(z)
imaginary = np.imag(z)

# plot the complex number on the complex plane
plt.plot((0,real),(0,imaginary),'ro--')

# some plotting touch-ups
plt.axis('square')
plt.axis([-5, 5, -5, 5])
plt.grid(True)
plt.xlabel('Real axis'), plt.ylabel('Imaginary axis')
plt.show()
```

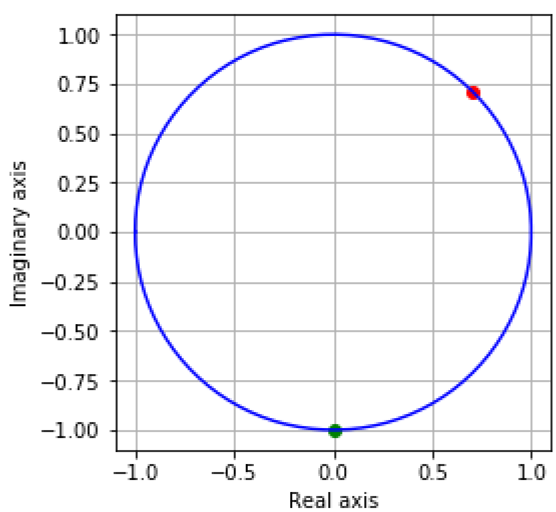


In [63]:
```
# compute the magnitude of the complex number using
# Pythagorean theorem
mag1 = np.sqrt(np.real(z)**2+np.imag(z)**2)
# or using abs
mag2 = np.abs(z)
# print out the magnitude of the complex number
print( 'The magnitude is',mag1,'or',mag2 )

# compute the angle of the complex number using trigonometry
phs1 = np.arctan(np.imag(z)/np.real(z))
# or using the angle function
phs2 = np.angle(z)
# print out the phase of the complex number
print( 'The angle is', phs2,'or',phs1 )
```

```
The magnitude is 5.0 or 5.0
The angle is 0.6435011087932844 or 0.6435011087932844
```

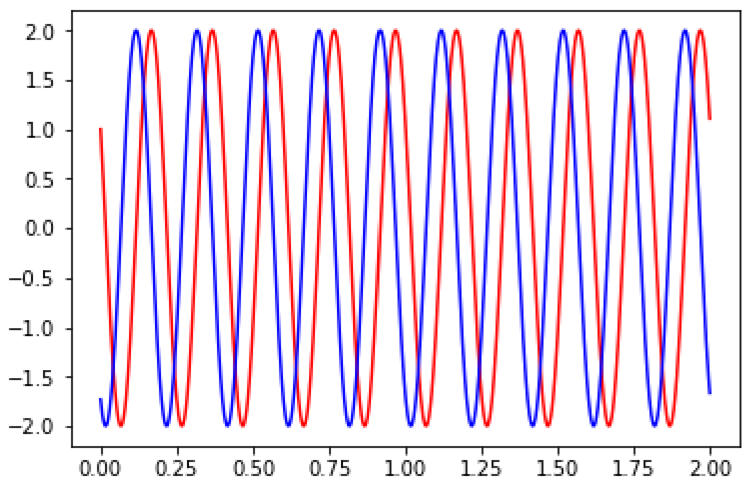
In [64]:
```
# define k (possibly an array of angles)
k = [np.pi/4,-1*np.pi/2]
print(k)

# Define the complex exponential here using Euler's formula
# (possibly with a lambda expression)

euler = lambda k : np.exp(k*1j)
print(euler(k[0]))

# plot dot

plt.plot(np.real(euler(k[0])),np.imag(euler(k[0])),'ro')
plt.plot(np.real(euler(k[1])),np.imag(euler(k[1])),'go')
#plt.plot(np.imag(euler(k)))
plt.xlabel('real axis')
plt.ylabel('imaginary axis')
# plt.xlim(-1.5,1.5)
# plt.ylim(-1.5,1.5)


# draw unit circle for reference
x=np.linspace(-np.pi,np.pi,100)
plt.plot(np.real(euler(x)),np.imag(euler(x)),'b')

#some plotting touch-ups
plt.axis('square')
plt.grid(True)
plt.xlabel('Real axis'), plt.ylabel('Imaginary axis')
plt.show()
```

```
[0.7853981633974483, -1.5707963267948966]
(0.7071067811865476+0.7071067811865476j)
```

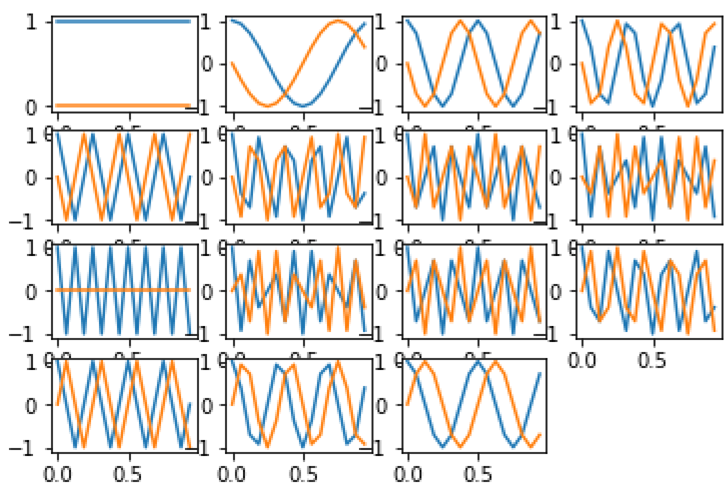


In [65]:
```
# complex sine waves

# general simulation parameters
srate =  500 # sampling rate in Hz
time  = np.arange(0,2,1/srate) # time in seconds
# sine wave parameters
freq = 5    # frequency in Hz
ampl = 2    # amplitude in a.u.
phase = np.pi/3 # phase in radians

# generate the sine wave
csw = ampl*np.exp(-1j*((2*np.pi*freq*time)+phase))
realsin = []

plt.plot(time,np.real(csw),'r')
plt.plot(time,np.imag(csw),'b')


# plot the results
# plt.plot(np.arange(0,time,srate),csw,'b')
```

Out[65]: [<matplotlib.lines.Line2D at 0x1fd8f5990f0>]



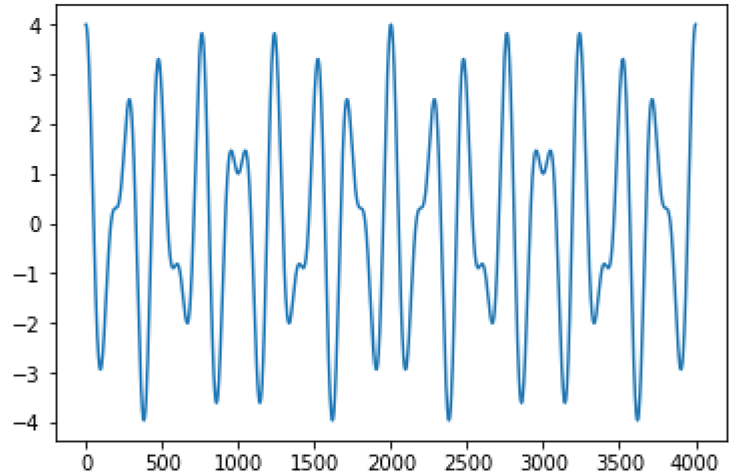
In [66]:
```
points = 16 # number of time points , sometimes denoted by N

# time vector to plot the basis
FourierTime = np.array(range(0,points))/points

# the slowest frequency in an N point sinusoid in Hz
slowest =  0
# the fastest frequency in an N point sinusoid in Hz
fastest = points-1

for fi in range(slowest,fastest):
    # create complex sine wave

    srate =  16 # sampling rate in Hz
    time  = FourierTime # time in seconds
    # sine wave parameters
    #freq = 5    # frequency in Hz
    ampl = 1    # amplitude in a.u.
    phase = 0 # phase in radians
    csw = ampl*np.exp(-1j*((2*np.pi*fi*time)+phase))

    # and plot it
    loc = np.unravel_index(fi,[4, 4],'F')
    plt.subplot2grid((4,4),(loc[1],loc[0]))
    plt.plot(FourierTime,np.real(csw))
    plt.plot(FourierTime,np.imag(csw))

plt.show()
```

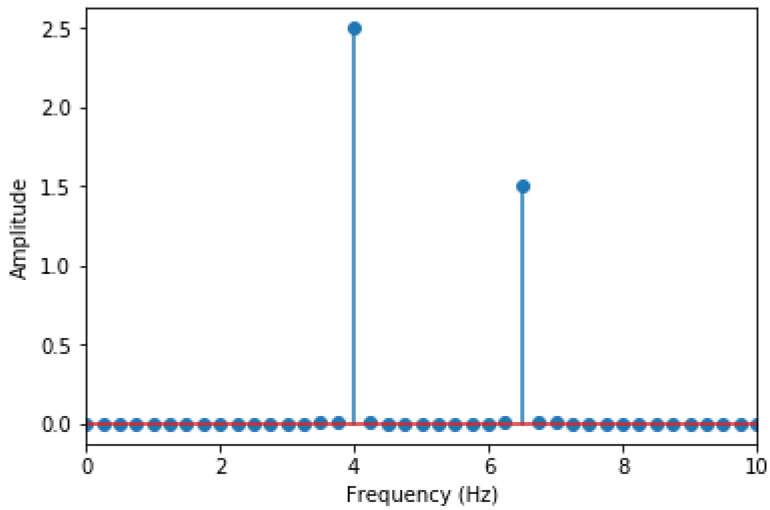


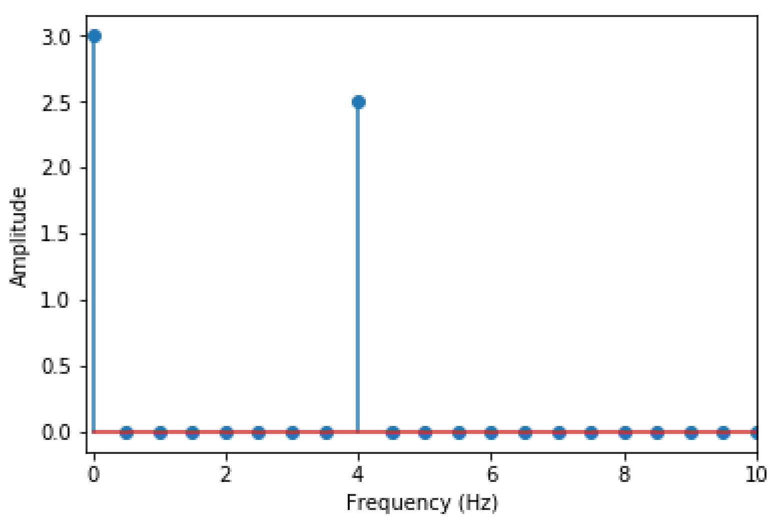
2) Naive Computation of the DFT and IDFT from First Principles (Vector Form)

In [67]:
```
## The DFT in loop-form

# create the signal 1
srate1 =  1000 # sampling rate in Hz
time1  = np.arange(0,4+1/(2*srate1),1/srate1) # time in seconds
freq1 = 4    # frequency in Hz
ampl1 = 2.5    # amplitude in a.u.
signal1 = ampl1*np.cos(2*np.pi*freq1*time1)



ampl2 = 1.5
freq2 = 6.5
# signal1 = signal1 + ampl*np.exp(-1j*((2*np.pi*freq*time)+phase))
signal1 = signal1 + ampl2*np.cos(2*np.pi*freq2*time1)
plt.plot(signal1)
plt.show()
pnts1=len(signal1)
# create the signal 2

srate2 =  1000 # sampling rate in Hz
time2  = np.arange(0,2,1/srate2) # time in seconds
freq3 = 4    # frequency in Hz
ampl3 = 2.5    # amplitude in a.u.


# signal2 = ampl*np.exp(-1j*((2*np.pi*freq*time)+phase))
signal2 = ampl3*np.cos((2*np.pi*freq3*time2))+1.5
pnts2 = len(signal2)

# prepare the Fourier transform for signal 1
fourTime1 = np.array(range(0,pnts1))/pnts1
fCoefs1  = np.zeros((len(signal1)),dtype=complex)

# prepare the Fourier transform for signal 2
fourTime2 = np.array(range(0,pnts2))/pnts2
fCoefs2   = np.zeros(len(signal2),dtype=complex)

for fi in range(0,pnts1):


    # compute dot product between sine wave and signal
    # these are called the Fourier coefficients
    csw1 = np.exp(-1j*2*np.pi*fi*fourTime1)
    fCoefs1[fi] = np.dot(csw1,signal1)/pnts1


for fi in range(0,pnts2):

    csw2 = np.exp(-1j*2*np.pi*fi*fourTime2)


    fCoefs2[fi] = np.sum(np.dot(csw2,signal2))/pnts2


# extract amplitudes for the spectrum of signal 1
```

```
ampls1 = 2*np.abs(fCoefs1)

# extract amplitudes for the spectrum of signal 2
ampls2 = 2*np.abs(fCoefs2);


# compute frequencies vector for the spectrum of signal 1
hz1 = np.linspace(0,srate1/2,num=math.floor(pnts1/2.)+1)

# compute frequencies vector for the spectrum of signal 2
hz2 = np.linspace(0,srate2/2,num=math.floor(pnts2/2.)+1)

fig1 = plt.figure(1)
plt.stem(hz1,ampls1[range(0,len(hz1))])
plt.xlabel('Frequency (Hz)'), plt.ylabel('Amplitude')
plt.xlim(0,10)

fig2 = plt.figure(2)
plt.stem(hz2,ampls2[0:len(hz2)])
plt.xlim(-.1,10)
plt.xlabel('Frequency (Hz)'), plt.ylabel('Amplitude')
plt.show()
```


4
3
2
1
0
−1
−2
−3
−4
0
500
1000
1500
2000
2500
3000
3500
4000

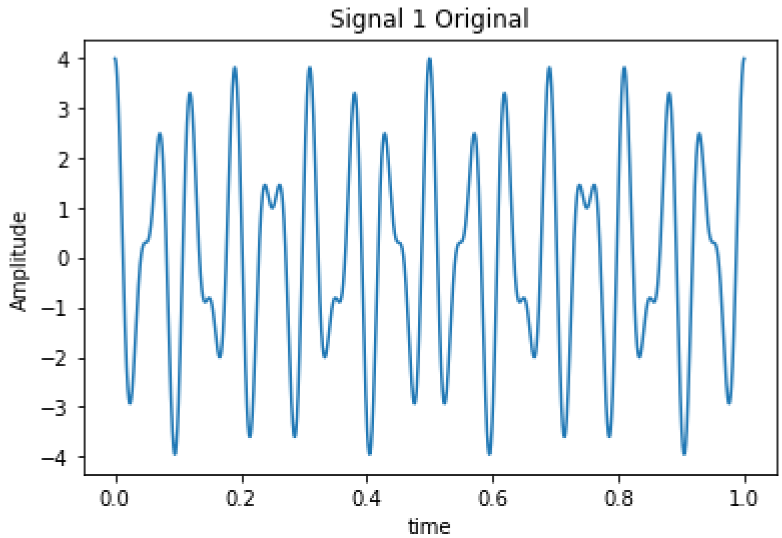

Amplitude
Frequency (Hz)
2.5
2.0
1.5
1.0
0.5
0.0
0
2
4
6
8
10

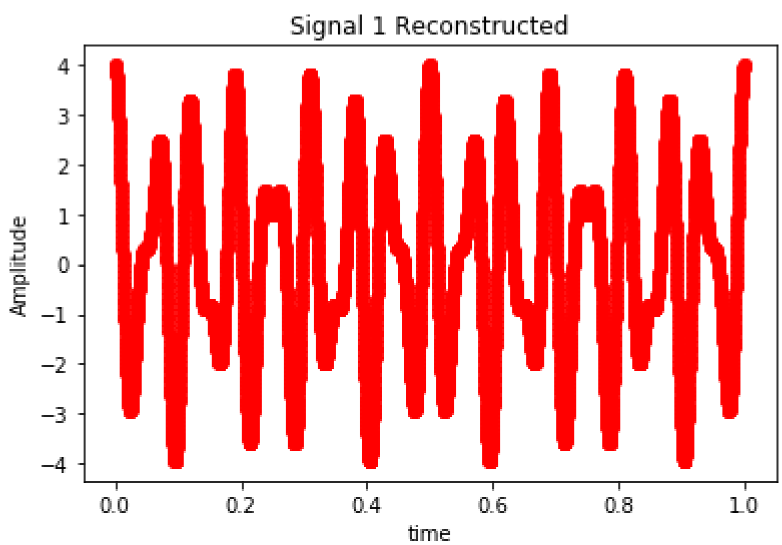

Amplitude
Frequency (Hz)
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
2
4
6
8
10

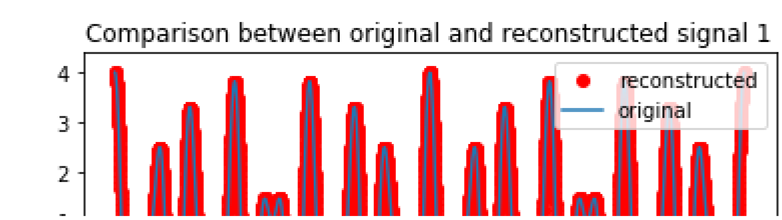
In [68]:
```
# IDFT (vector)

# initialize time-domain reconstruction for signal 1
reconSignal1 = np.zeros((len(signal1)),dtype=complex)
# initialize time-domain reconstruction for signal 2
reconSignal2 = np.zeros((len(signal2)),dtype=complex)


for fi in range(0,pnts1):

    # create coefficient-modulated complex sine wave
    inv_csw1 = np.exp(1j*2*np.pi*fi*fourTime1) * fCoefs1[fi]

    # sum them together
    reconSignal1 += inv_csw1

for fi in range(0,pnts2):

    # create coefficient-modulated complex sine wave
    inv_csw2 = np.exp(1j*2*np.pi*fi*fourTime2) * fCoefs2[fi]

    # sum them together
    reconSignal2 += inv_csw2

#plot the results for signal 1
fig1 = plt.figure(1)
fig1.set_figheight(20)
fig1.subplots_adjust(hspace=1, wspace=1, left=0.1)

plt.subplot(3,1,1)
plt.plot(fourTime1, signal1)
plt.xlabel('time')
plt.ylabel('Amplitude')
plt.title('Signal 1 Original')

plt.subplot(3,1,2)
plt.plot(fourTime1, reconSignal1, 'ro')
plt.xlabel('time')
plt.ylabel('Amplitude')
plt.title('Signal 1 Reconstructed')

plt.subplot(3,1,3)
plt.plot(fourTime1, reconSignal1, 'ro', label='reconstructed')
plt.plot(fourTime1, signal1, label='original')
plt.title('Comparison between original and reconstructed signal 1')
plt.legend()

#plot the results for signal 2
fig1 = plt.figure(2)
fig1.set_figheight(20)
fig1.subplots_adjust(hspace=1, wspace=1, left=0.1)

plt.subplot(3,1,1)
plt.plot(fourTime2, signal2)
plt.xlabel('time')
plt.ylabel('Amplitude')
```

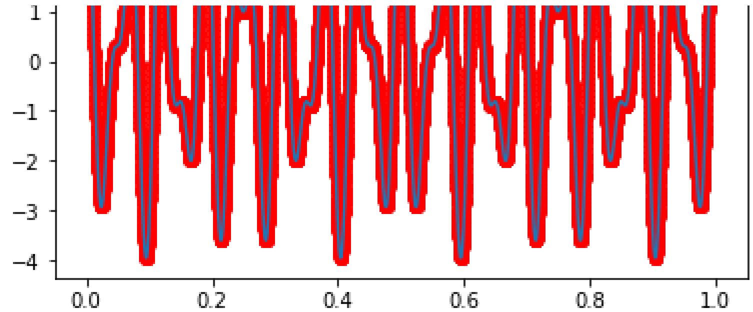
```python
plt.title('Signal 2 Original')

plt.subplot(3,1,2)
plt.plot(fourTime2, reconSignal2, 'ro')
plt.xlabel('time')
plt.ylabel('Amplitude')
plt.title('Signal 2 Reconstructed')

plt.subplot(3,1,3)
plt.plot(fourTime2, reconSignal2, 'ro', label='reconstructed')
plt.plot(fourTime2, signal2, label='original')
plt.title('Comparison between original and reconstructed signal 2')
plt.legend()
```

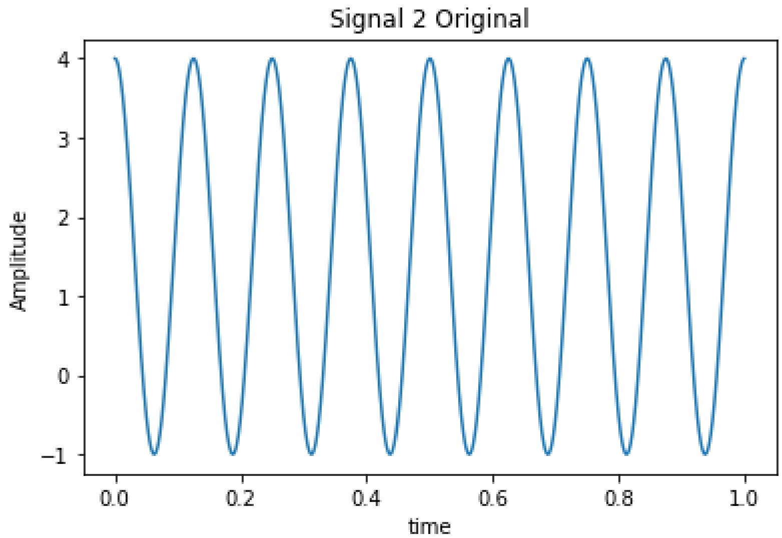
```
Out[68]: <matplotlib.legend.Legend at 0x1fd9f1d87f0>
```

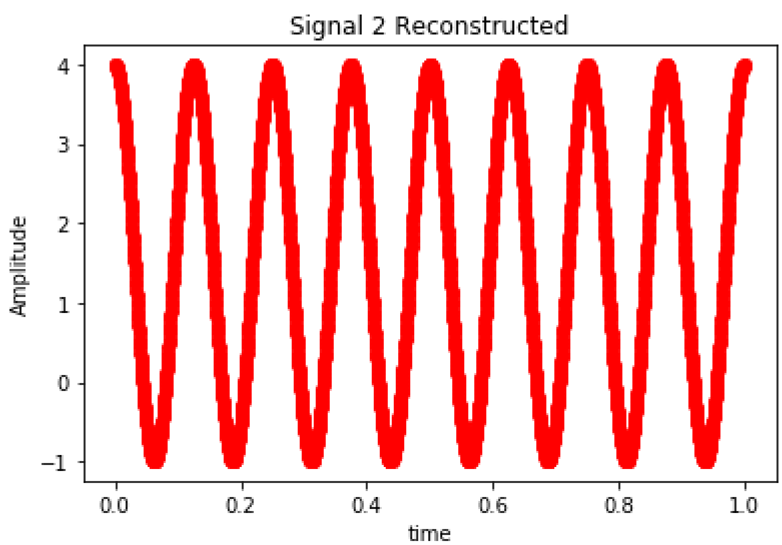

Signal 1 Original
Amplitude
time

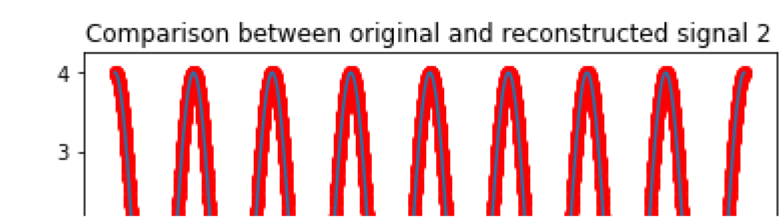

Signal 1 Reconstructed
Amplitude
time

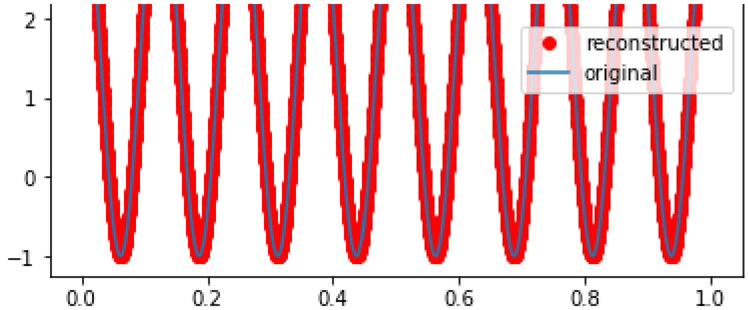

Comparison between original and reconstructed signal 1
reconstructed
original


1
0
−1
−2
−3
−4
0.0
0.2
0.4
0.6
0.8
1.0


Signal 2 Original
Amplitude
time
4
3
2
1
0
−1
0.0
0.2
0.4
0.6
0.8
1.0


Signal 2 Reconstructed
Amplitude
time
4
3
2
1
0
−1
0.0
0.2
0.4
0.6
0.8
1.0


Comparison between original and reconstructed signal 2
4
3



3) Naive Computation of the DFT and IDFT from First Principles (Matrix Form)

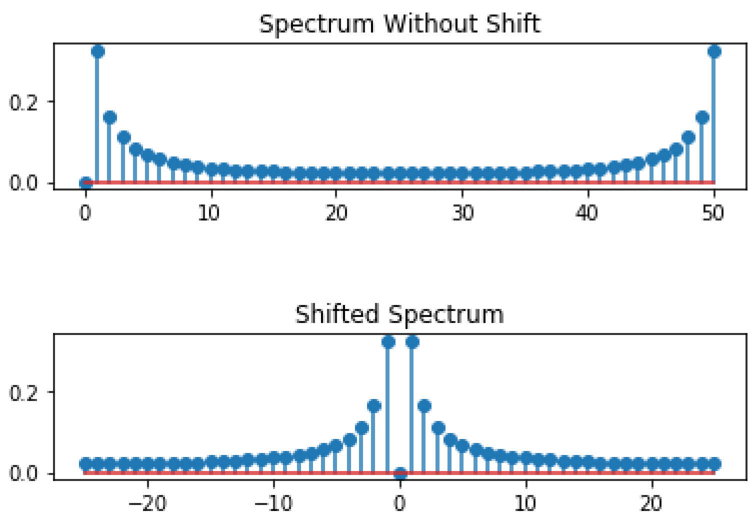
In [69]:
```
def dft_matrix(N):
    # create a 1xN matrix containing indices 0 to N-1


    # take advantage of numpy broadcasting to create the matrix

    ## OR

    # use a nested for loop to populate an NxN matrix
    W = np.zeros((N,N), dtype = complex)
    for n in range(N):
        for k in range(N):
            W[n][k] = np.exp(1j*np.pi*2*n*k/N)
    return W
```

In [70]:
```
def dft(signal,N):

    # Obtain DFT matrix for signal
    W = dft_matrix(N)
    # Find the DFT for signal
    X = ((np.matmul(W,signal))) / N

    # return the DFT

    return X
```

In [71]:
```
def dft_shift(X):
    N = int(len(X))
    if (N % 2 == 0):
        # even-length: return N+1 values
         # specify the range of frequency bins in the DFT
        n = np.arange(-N/2,N/2 + 1)
        # create the shifted spectrum
        Y = np.concatenate((X[int(N/2)], X[int(N/2):N], X[0:int(N/2)]), axis =
None)
        return n,Y
    else:
        # odd-length: return N values

        # specify the range of frequency bins in the DFT
        n = np.arange(-(N-1)/2,(N-1)/2 + 1)
        # create the shifted spectrum
        shift = ((N-1)/2) + 1
        Y = np.concatenate((X[int(shift):int(N)], X[0:int(shift)]), axis = Non
e)
        return n,Y
```

In [72]:
```
# test your shift function here

x = np.arange(0, 1.02, 0.02) - 0.5 # test signal
X = dft(x,len(x)) # obtain DFT of the test signal

fig_test=plt.figure(100)
fig_test.subplots_adjust(hspace=1, wspace=1, left = 0.1)
# plot the spectrum without shift
plt.subplot(2,1,1)
plt.stem(2*abs(X));
plt.title('Spectrum Without Shift')

n, y = dft_shift(X) # obtain shifted spectrum
# plot the shifted spectrum
plt.subplot(2,1,2)
plt.stem(n, 2*abs(y));
plt.title('Shifted Spectrum')
```

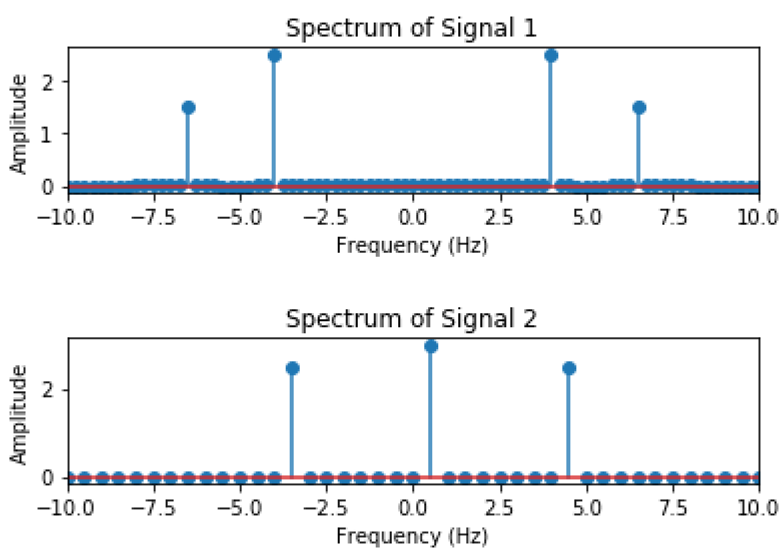
Out[72]:
```
Text(0.5, 1.0, 'Shifted Spectrum')
```



In [73]:
```
def dft_map(X, Fs, shift):
    # define the resolution
    resolution =  float(Fs) / len(X)

    if shift:
        # apply a shift if the condition is True

        # get both the frequency bins and the shifted spectrum
        n, Y =  dft_shift(X)
    else:
        Y = X # retain the original spectrum for no shift

        # the range of frequency bins is from 0 to
        # the length of the signal for no shift
        n = np.arange(0, len(Y))

    f =  n * resolution# obtain frequency vector

    return f, Y
```

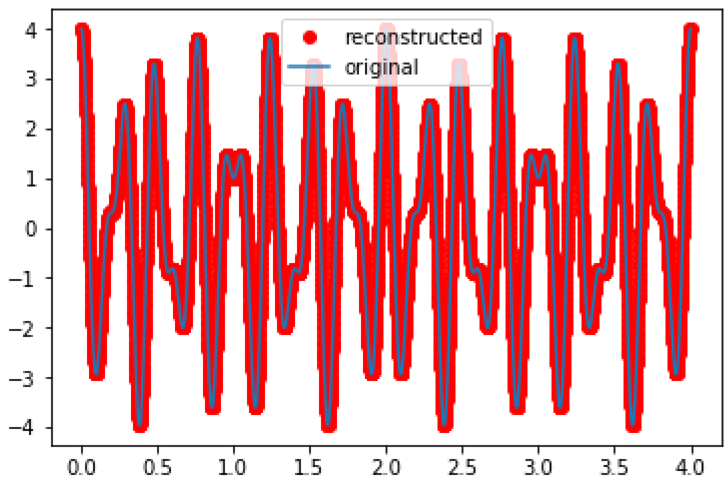
In [74]:
```python
# Find the DFT for signal 1
X1 = dft(signal1, len(signal1))

f1, X1Y = dft_map(X1, 1000, 1)
# obtain absolute value
absX1Y = 2*abs(X1Y)
# plot the result
fig4 = plt.figure(4)
fig4.subplots_adjust(hspace= 1, wspace= 1, left = 0.1)
plt.subplot(2,1,1)
plt.title('Spectrum of Signal 1')
plt.xlabel('Frequency (Hz)'), plt.ylabel('Amplitude')
plt.stem(f1, absX1Y)
plt.xlim(-10, 10)

# Find the DFT for signal 2
X2 = dft(signal2, len(signal2))
f2, X2Y = dft_map(X2, 1000, 1)
# obtain absolute value
absX2Y = 2*abs(X2Y)
# plot the result
plt.subplot(2,1,2)
plt.title('Spectrum of Signal 2')
plt.xlabel('Frequency (Hz)'), plt.ylabel('Amplitude')
plt.stem(f2, absX2Y)
plt.xlim(-10, 10)
```

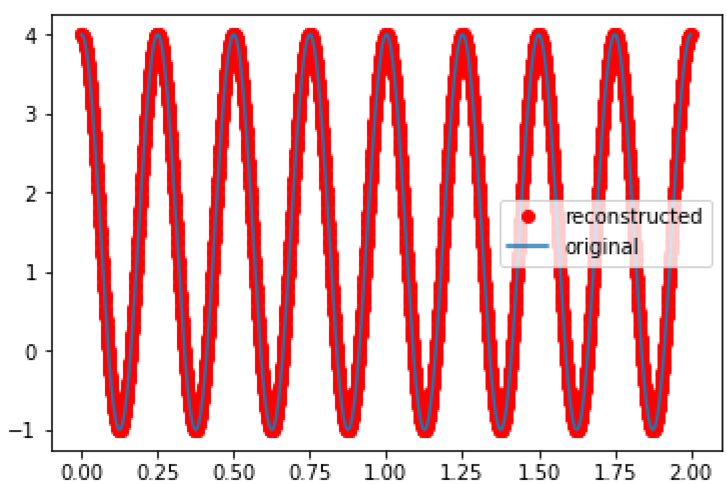
Out[74]: (-10, 10)



In [75]:
```
# IDFT (matrix)

# Obtain the DFT matrix
W1 = dft_matrix(pnts1)
# inverse DFT fo signal 1
x_hat1 = np.matmul(W1.T.conjugate(), X1)
# plot the result
fig5 = plt.figure(5)
plt.plot(time1, x_hat1, 'ro', label='reconstructed')
plt.plot(time1, signal1, label='original')
plt.legend()

# Obtain the DFT matrix
W2 = dft_matrix(pnts2)
# inverse DFT for signal 2
x_hat2 = np.matmul(W2.T.conjugate(), X2)
# plot the result
fig6 = plt.figure(6)
plt.plot(time2, x_hat2, 'ro', label='reconstructed')
plt.plot(time2, signal2, label='original')
plt.legend()
```

Out[75]: <matplotlib.legend.Legend at 0x1fdb7cc9b70>



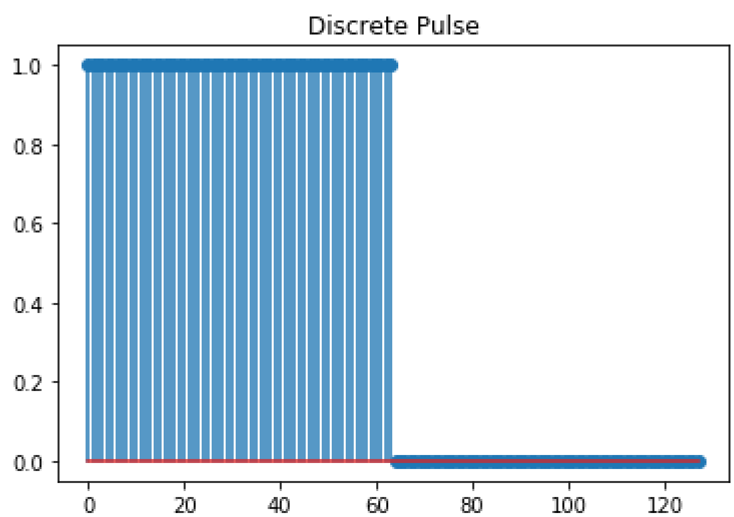


4) Numerical Precision Issues With the DFT and IDFT

In [83]:
```
# define a discrete step function
def u(n):
    return 1 * (n >= 0.0)
```

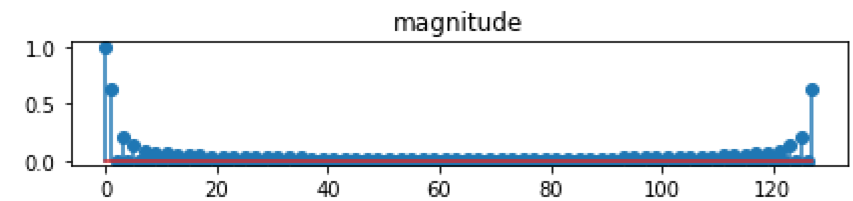
In [84]:
```
N = 128 # define the number of points in the discrete time pulse
n =  np.arange(0, N) # discrete time index values

pulse = u(n) - u(n-64)  # obtain the discrete pulse
# plot the pulse
plt.title('Discrete Pulse')
plt.stem(n, pulse)
```

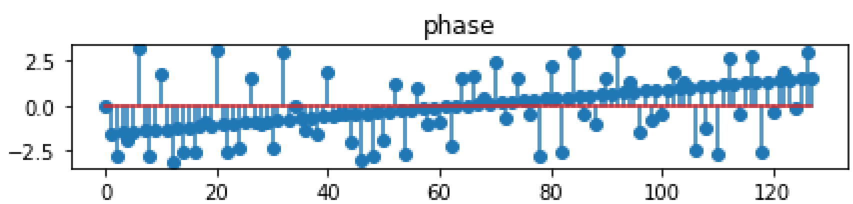
Out[84]: <StemContainer object of 3 artists>



In [89]:
```
P = dft(pulse, N) # obtain the DFT of the pulse using your DFT function


# plot the magnitude and phase of the pulse's spectrum
fig7 = plt.figure(7)
fig7.subplots_adjust(hspace=1.5, wspace=1, left = 0.001)

plt.subplot(2,1,1)
plt.title('magnitude')
plt.stem(n, np.abs(P) * 2)

plt.subplot(2,1,2)
plt.title('phase')
plt.stem(n, -1 * np.angle(P))
```

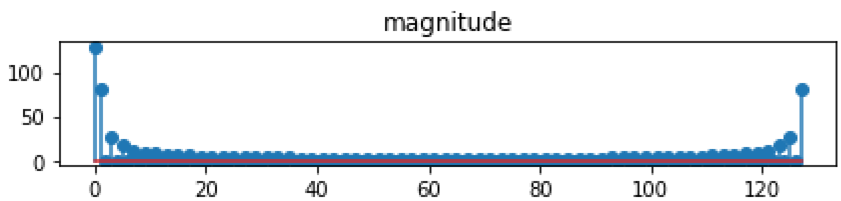
Out[89]: <StemContainer object of 3 artists>

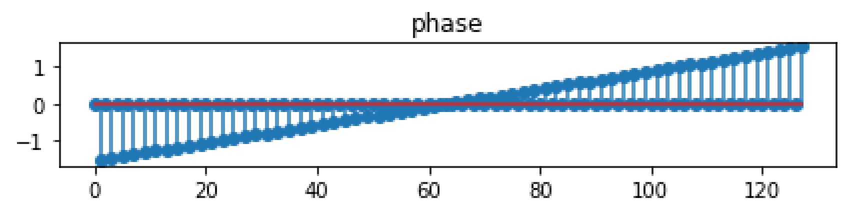




In [90]:
```
# Obtain the DFT using scipy or numpy's fft function
P2 =  fft(pulse, N)

# plot the magnitude and phase of the pulse's spectrum
fig8 = plt.figure(8)
fig8.subplots_adjust(hspace= 1.5, wspace= 1, left = 0.001)

plt.subplot(2,1,1)
plt.title('magnitude')
plt.stem(n, np.abs(P2) * 2)

plt.subplot(2,1,2)
plt.title('phase')
plt.stem(n, np.angle(P2))
```

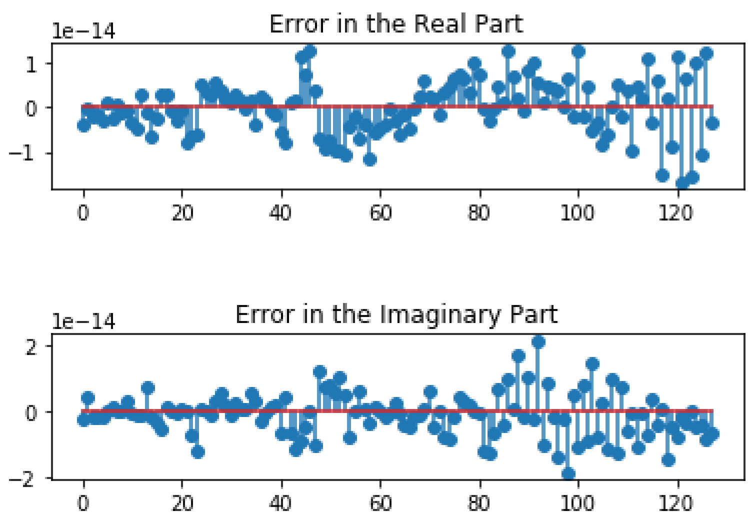
Out[90]: <StemContainer object of 3 artists>





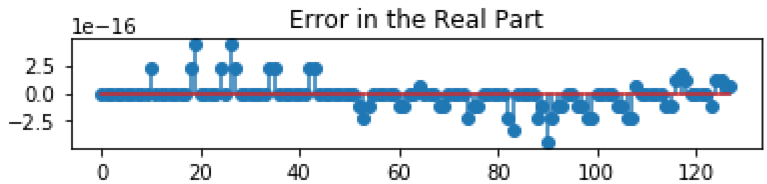
In [96]:
```
# Obtain the N point DFT matrix
W = dft_matrix(N)
# use the IDFT to obtain the reconstructed time-domain signal
x_hat =  np.matmul(W.T.conjugate(), P)

# plot the error (for both, imaginary and real parts)
# between the original signal and the reconstructed signal
fig9 = plt.figure(9)
fig9.subplots_adjust(hspace=1, wspace=1, left = 0.1)

plt.subplot(2,1,1)
plt.title('Error in the Real Part')
plt.stem(n, (np.real(x_hat) - np.real(pulse)))


plt.subplot(2,1,2)
plt.title('Error in the Imaginary Part')
plt.stem(n, (np.imag(x_hat) - np.imag(pulse)))
```

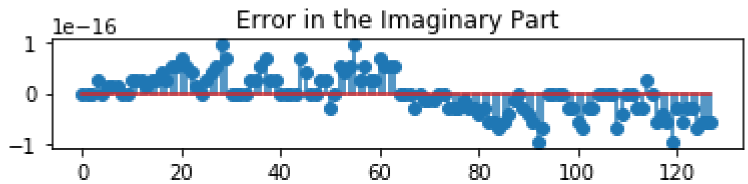
Out[96]: <StemContainer object of 3 artists>



In [97]:
```
# use the IDFT function in scipy or numpy
# to obtain the reconstructed time-domain signal
x_hat = ifft(P2)

# plot the error (for both, imaginary and real parts)
# between the original signal and the reconstructed signal

fig10 = plt.figure(10)
fig10.subplots_adjust(hspace=2, wspace=1, left = 0.1)

plt.subplot(2,1,1)
plt.stem(n, np.real(x_hat) - np.real(pulse))
plt.title('Error in the Real Part')

plt.subplot(2,1,2)

plt.stem(n, -1 * (np.imag(x_hat) - np.imag(pulse))/2)
plt.title('Error in the Imaginary Part')
```

Out[97]: `Text(0.5, 1.0, 'Error in the Imaginary Part')`





5) Minimizing Energy Spread and Zero Padding

In [ ]:
```
def minimizeEnergySpreadDFT(x, fs, f1, f2):
    ''''
    Inputs:
    x : signal
    fs : sampling rate
    f1 : frequency of one of the sinusoids
    f2 : frequency of the other sinusoid

    Outputs:
    mX : the spectrum of x (with shift if needed) with minimum
    spectral lekage
    f: the corresponding frequency vector
    ''''

    t1 =  # time period of the discrete time or sampled signal
    t2 =  # time period of the discrete time or sampled signal
    M =  # LCM of the two periods

    X =  # M point FFT of the signal

    # obtain the frequency mapping and shifted spectrum
    f,mX =

    return mX,f
```

In [ ]:
```
#Define the sampling rate and the signal


# Plot the DFT after minimizing the energy spread or spectral
# leakage



# Plot the DFT before minimizing the energy spread or spectral
# leakage
```

In [ ]:
```
def optimalZeropad(x, fs, f):

    M =  # store the length of the signal

    # calculate the number of zeros to be padded
    period_samples =
    fraction =
    pad =

    N =  # find the length of the signal after zero padding

    x =  # pad the signal with zeros

    X =  # obtain the DFT of the zero padded signal

    # obtain the frequency mapping and shifted spectrum
    f,mX =

    return mX,f
```

In [ ]:
```
# Define the sampling rate and the signal


# Find DFT without zero padding and plot the result


# Find DFT after zero padding and plot the result
```

In [ ]:
```
N = 256 # Number of points
Delta =  # write down the expression for Delta
n =  # obtain the discrete time vector

omega =  # define the main frequency

# construct a signal that is composed of two cosine waves that are
# well more than Delta apart in the frequency domain
x =

# find the DFT and plot the first half of the magnitude spectrum
```

In [ ]:
```
# construct a signal that is composed of two cosine waves that are
# less than Delta apart in the frequency domain
x =
# find the DFT and plot the first half of the magnitude spectrum
```

In [ ]:
```
# create a zero padded version of the signal that has cosines less
# than Delta apart
xzp =
# find the DFT and plot the first half of the magnitude spectrum
```