Michael Christensen
CS 450
October 24, 2013

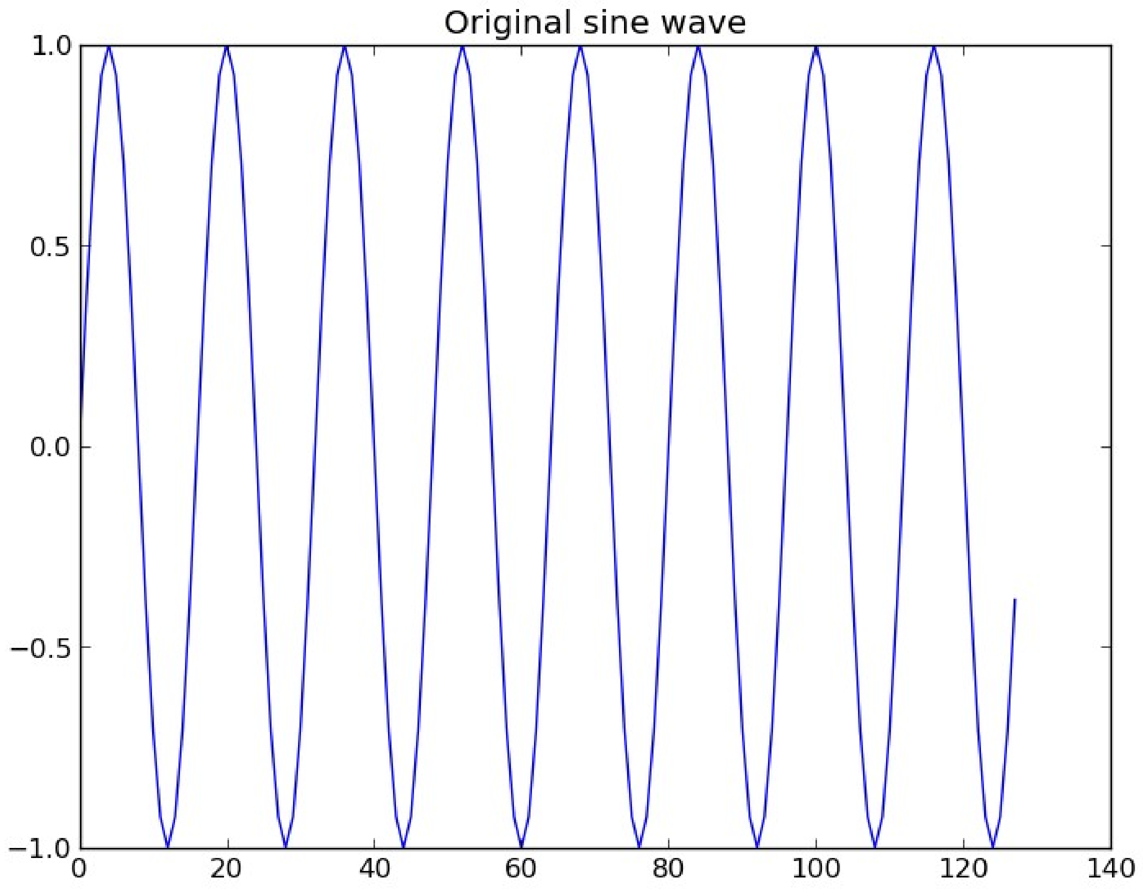
Homework 4 -- Programming Part

Part A (DFT code):

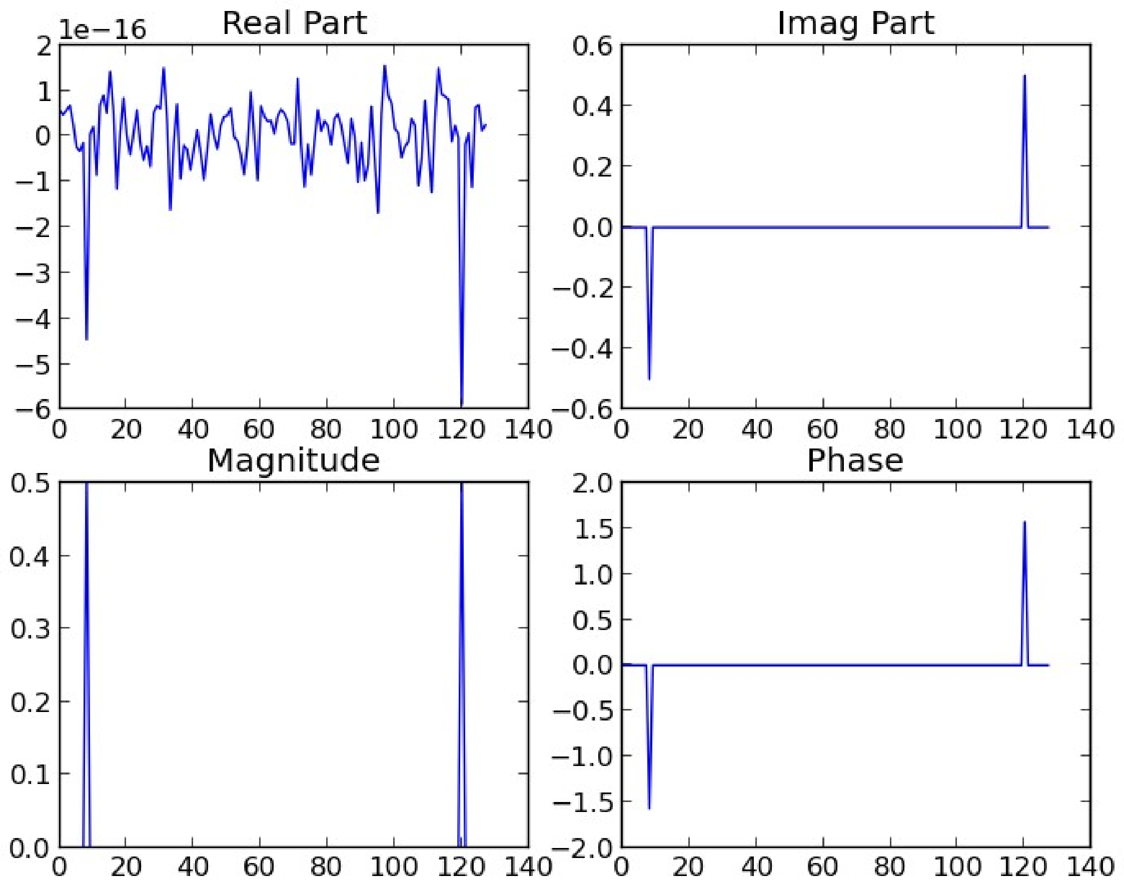
```
def dft_enhanced(signal_arr):
    siglen = len(signal_arr)
    # tabulate values of cos(2pik/M) and sin(2pik/M)
    cosarr = list(np.cos(2 * np.pi * k / siglen) for k in range(siglen))
    sinarr = list(np.sin(2 * np.pi * k / siglen) for k in range(siglen))

    Fu = np.zeros(siglen, dtype=np.complex)
    for u in range(Fu.size):
        Fu[u] = np.complex(0, 0)
        for x in range(Fu.size):
            k = (u * x) % siglen
            Fu[u] += np.complex(signal_arr[x] * cosarr[k], \
                        signal_arr[x] * -sinarr[k])
        Fu[u] /= siglen
    return Fu
```

Part B: Simple sines and cosines:

1. Sine wave f[t] = sin(2 pi s t / N) where s = 8 and N = 128

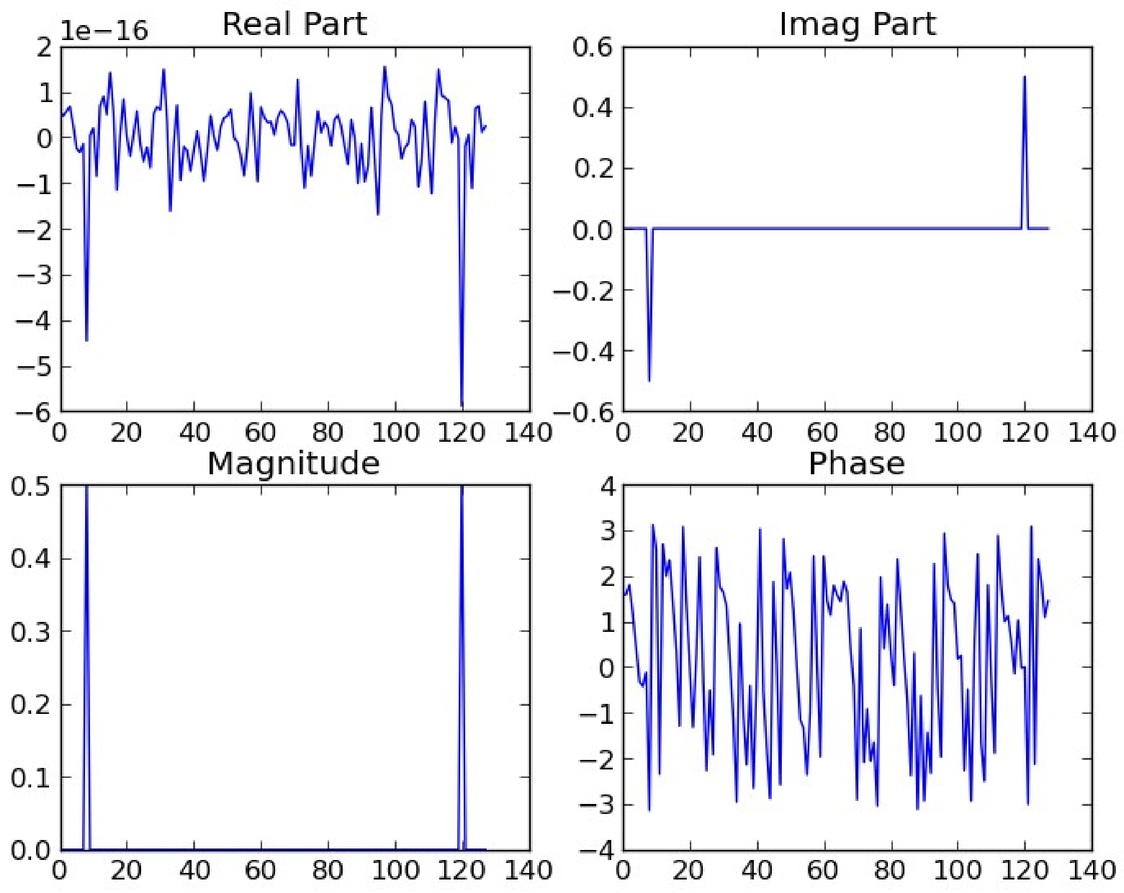


Parts of the wave:

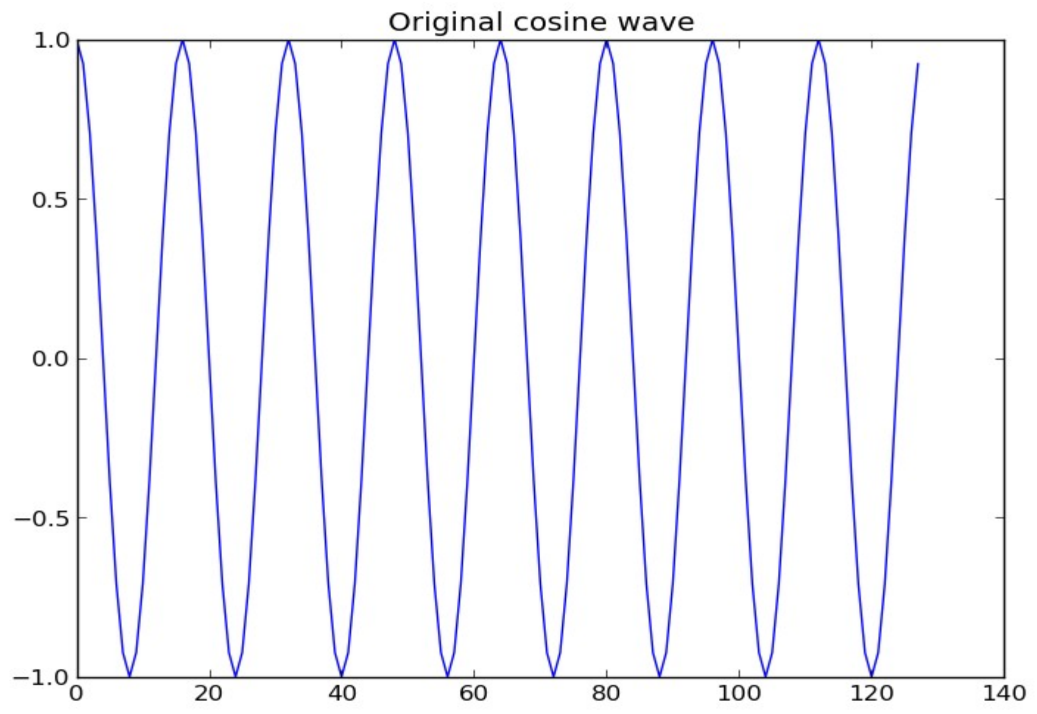


Explanation of each part of the figure above:
As a note, in all of the graphs above, they have not been shifted to be centered on the origin nor wrapper around (I.e. the rightmost peak in the “Imag Part” figure located at 20 on the x-axis should really be located at -8 (just to the left of the origin). In the “Real Part” figure, every value is roughly zero (notice the “1e-16” at the top of the y-axis), showing that the original sine wave is made up of zero un-shifted sinusoids (i.e. cosine waves). In the “Imag Part” figure, on the other hand, you see two peaks at *frequency u* = 8 and 120(which we can think of as -8 per my explanation in the first sentence), where *F(8)* = -.5 and *F(-8)* = 0.5. The 8 refers to the frequency *s* given in the original wave (and since sine is an odd function, we also get the -8), meaning that the original signal was made up of a sinusoid of frequency 8 (or -8), and nothing in the real part, which makes sense since the original signal was a pure sine wave and not the compilation of many waves of different frequencies/amplitudes/phase shifts. The magnitude simply corresponds to the coefficient of *i* in the complex number representation of this wave (notice how the magnitude at *u* = 8 and *u* = -8 in the “Magnitude” figure is equal to the value at those locations in the “Imag Part” figure, since the square root of *a* squared plus *b* squared equals the square root of 0 + *b* squared, which is just the original *b*. In the “Phase” figure, the numbers should be zero when *a* and *b* (from the wave representation *a* + *b*) equal zero (since phase = arctan(*b*/*a)* according to the complex plane). Since there is no cosine part of this wave, *a* = 0, and so the phase graph should be roughly zero as well (arctan(*b*/0) → 0). At 8 and -8, `numpy.angle(complex_number)`returns +-1.57 rad, which is +-90 degrees, meaning the complex number is entirely imaginary (in the real/imaginary plane, all along the imaginary axis). In a previous attempt at this, I had fluctuations in the phase graph

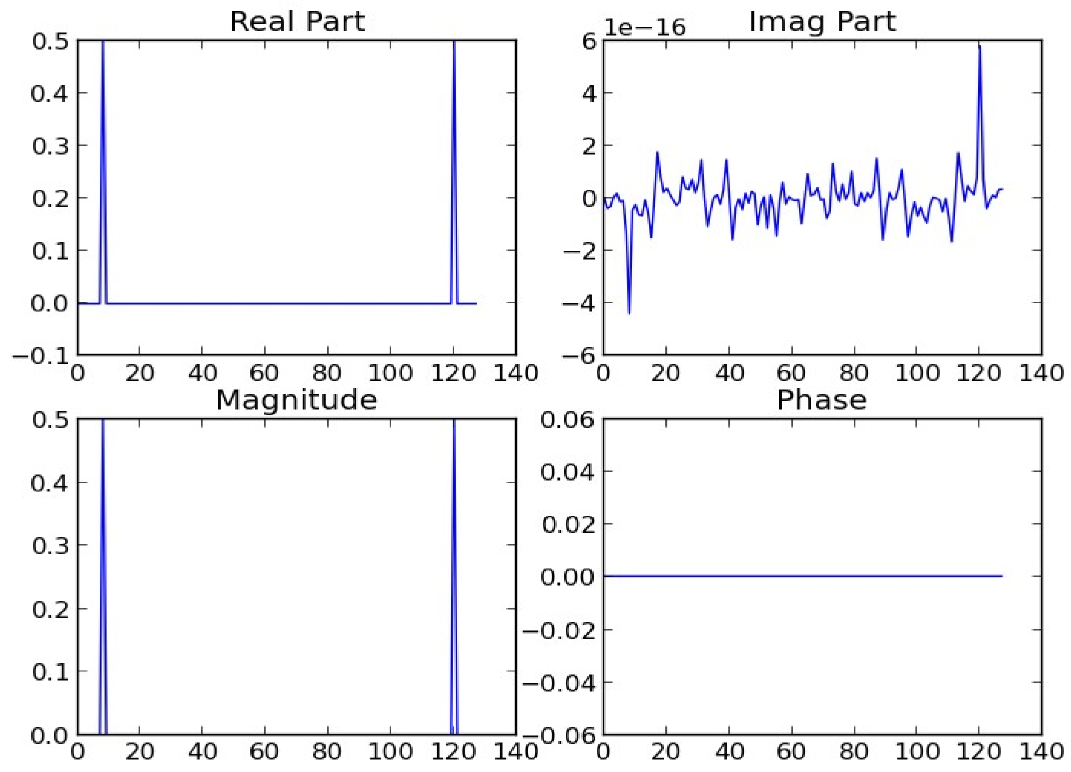
as a result of numbers being divided by small but non-zero numbers (like 1e-16). To alleviate this, I filtered the results of the Fourier transform by making every complex number near 0 equal to zero. The previous results of which I am referring are here:



2. Cosine wave f[t] = cos(2 pi s t / N) where s = 8 and N = 128

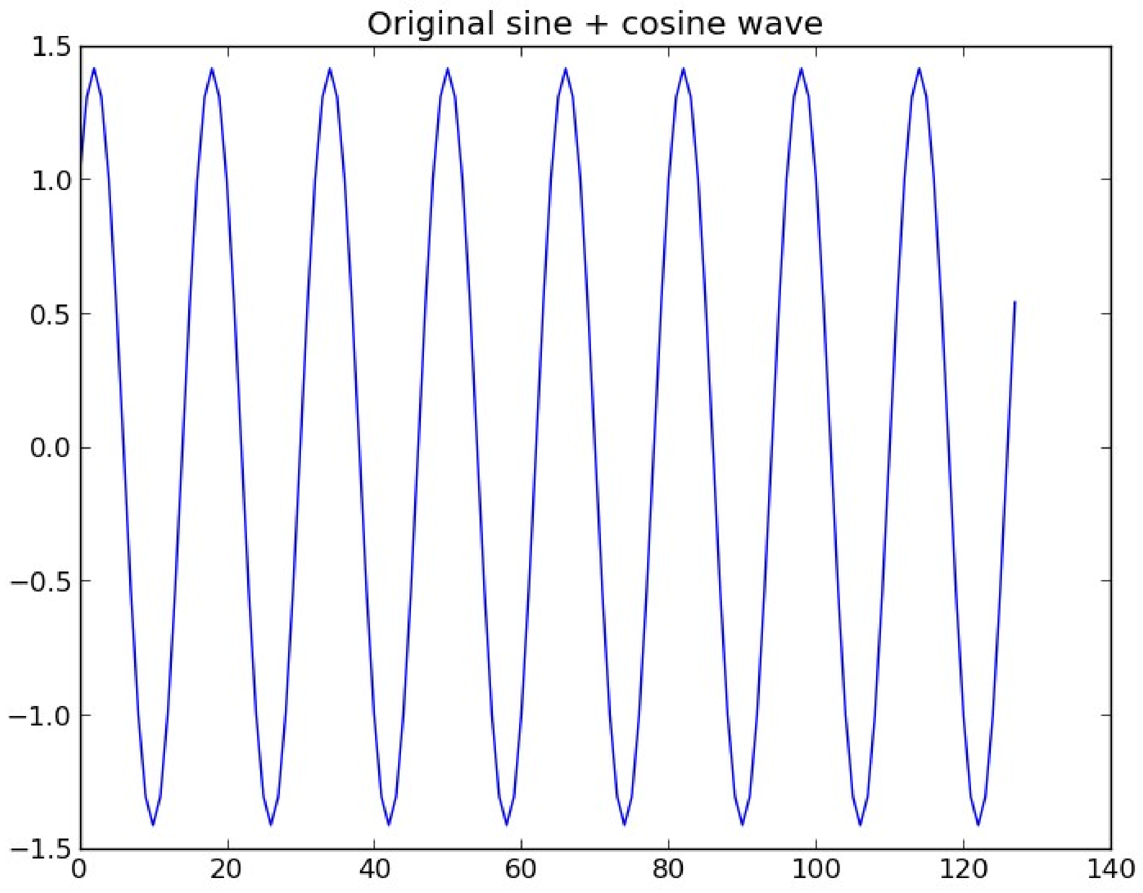


Parts of the wave:

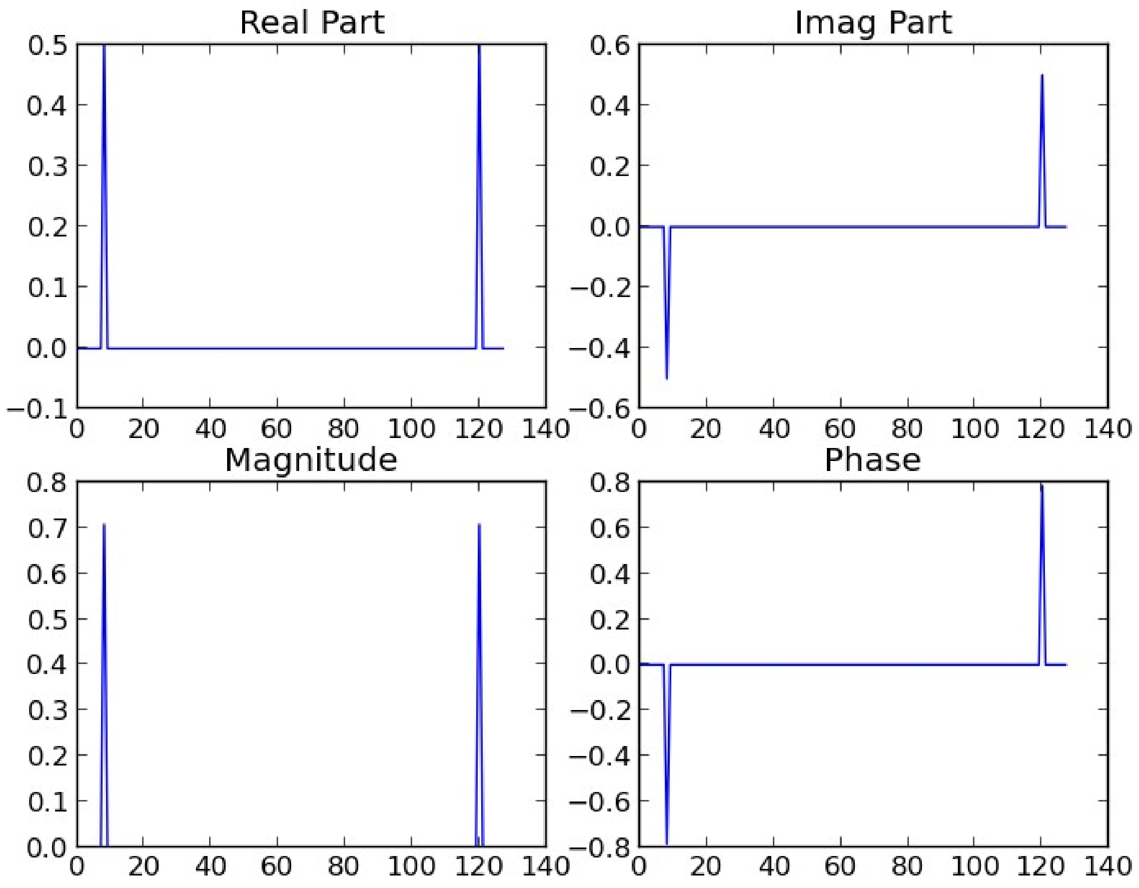


Explanation of parts of the wave:
The explanation of these parts will be very similar to the explanation in part 1 of this question (about the sine wave). Instead of spikes in the imaginary part, there are spikes in the real part corresponding to the frequency value of 8, except since cosine is an even function the value corresponding to a frequency of -8 is positive. The imaginary part of the wave is 0 (roughly) everywhere, as expected, since the original sinusoid is a pure cosine wave and thus isn't made up of any sine waves (the sinusoids that correspond to -*i*sin(theta) in Euler's equation). The magnitude simply correlates to the real part of the signal, since there is no imaginary part; you can consider the magnitude here and elsewhere in the document as the amplitude of the combined cosine and sine waves that make up the original signal). The phase graph is completely zero since this corresponds to the complex number being entirely real and therefore a point along the real axis (which translates to an angle of 0 or pi). This means there is no phase shift in the sinusoid, which is expected because it is a pure cosine wave.

3. Sum of the sine and cosine functions:

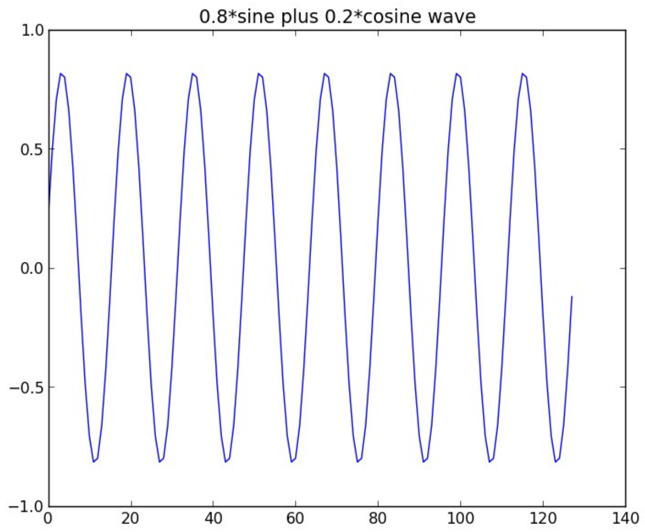


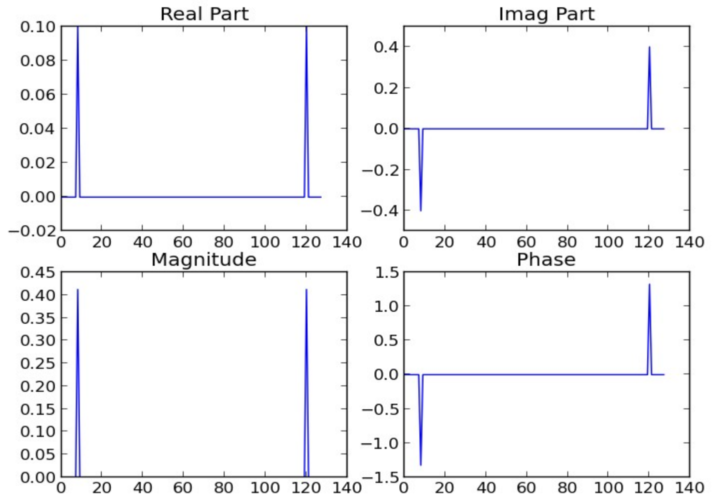
Parts of the combination wave:

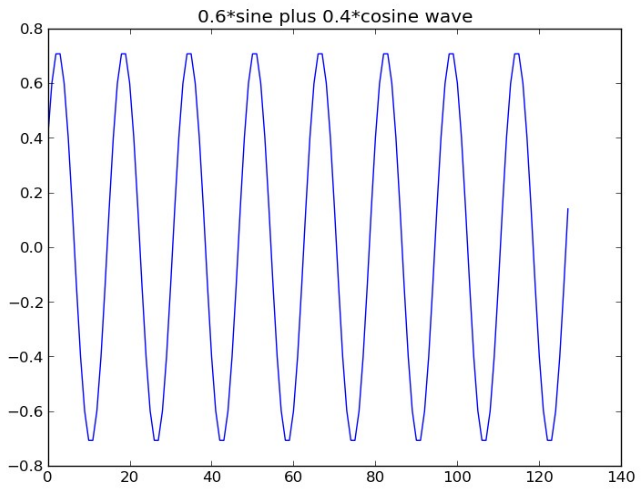


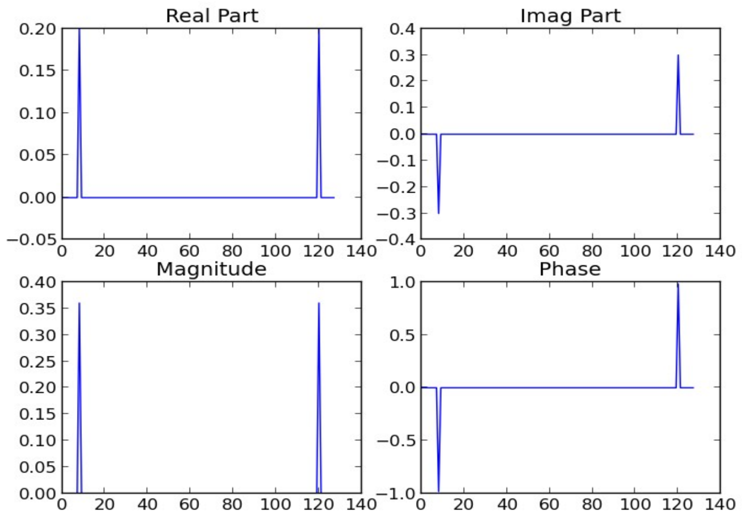
Explanation of parts of the wave:
Since this original wave is a combination of sine and cosine waves, it is expected that both the “Real” and “Imag Part” figures in the graphic above show non-zero values at the appropriate frequency value (plus/minus 8). The magnitude spikes are representative of the combination of the cosine and sine signals, since 0.5 + 0.5i → squareroot(0.5^2 + 0.5^2) = 0.707, as shown on the graph. Thus, at frequency y = 8, the amplitude of the transformed signal is 0.707. The phase graph is zero everywhere except at frequency values 8 and -8, and at those specific values corresponding to a frequency of 8, it shows that the phase is equal to +- 0.785 rad, or 45 degrees. This makes sense since the original wave is a combination of equal parts sine and cosine waves, so the complex number representation reflects this by having an angle equally between both the real and imaginary axis.

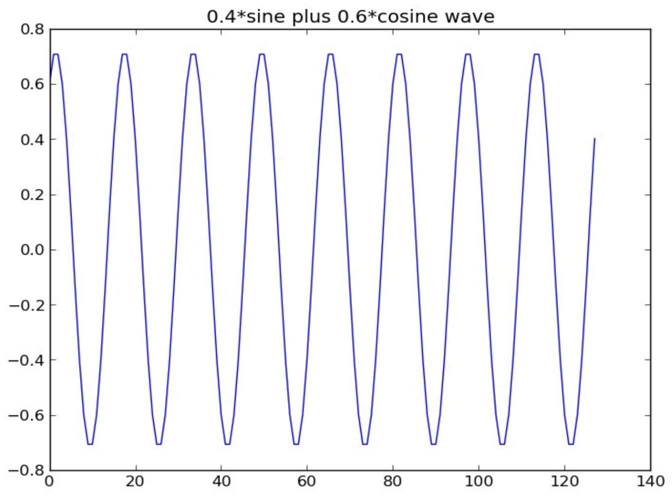
4. Playing with relative weightings of the sine and cosine parts:

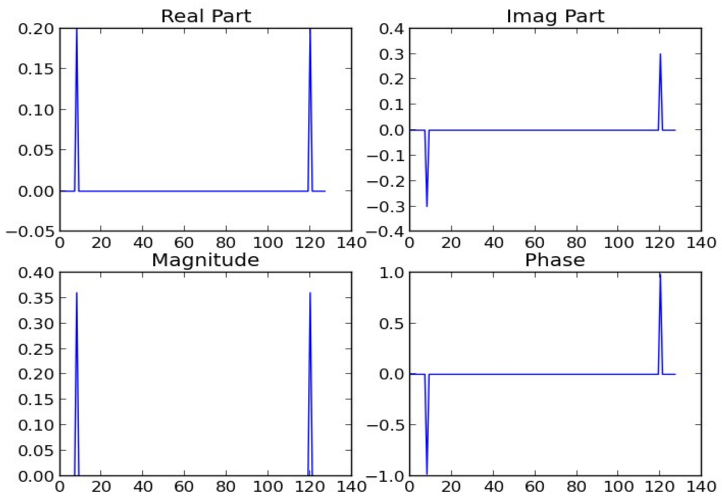


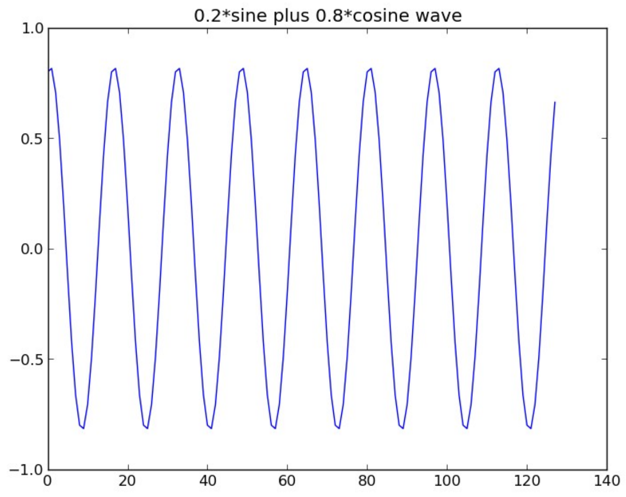


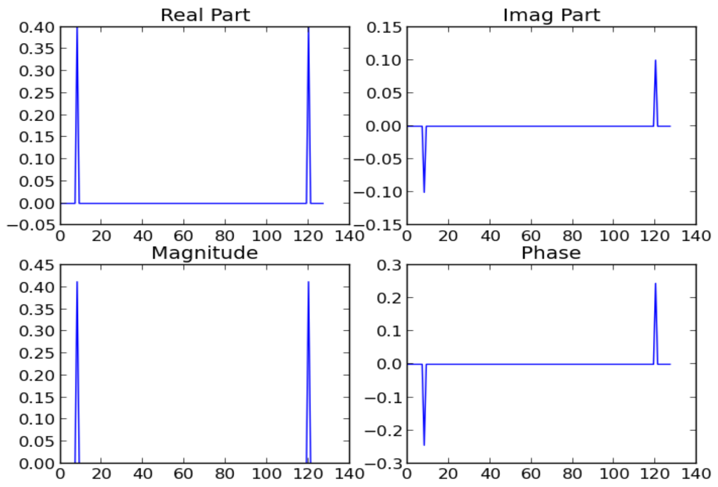








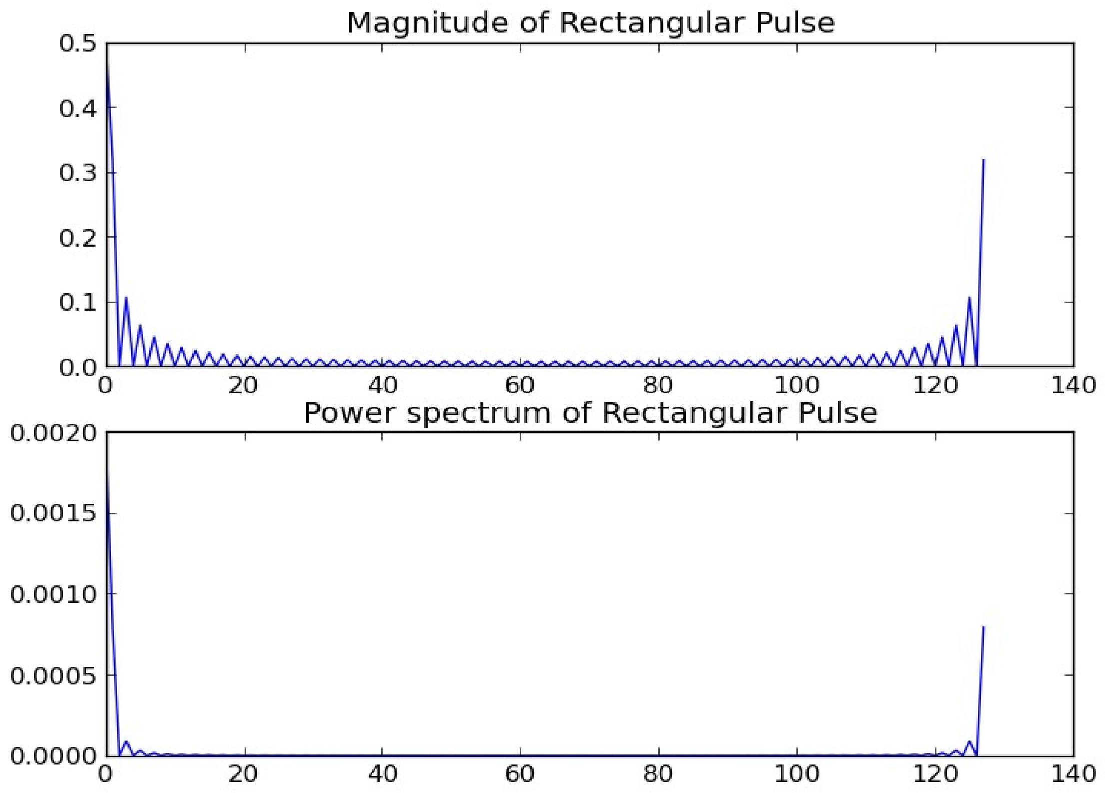




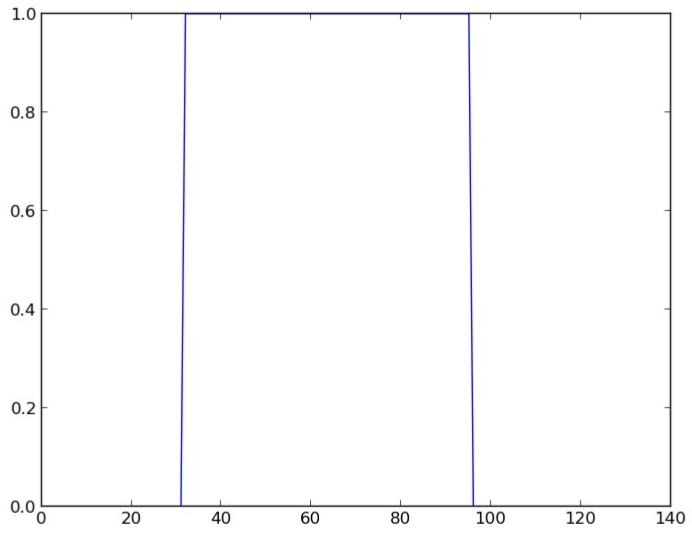
Report of my findings:
The graphs appear and can be explained in the same way as in part 3 previously. The interesting thing to notice is that change in phase values. The highest phase value corresponds to where the values of the imaginary and real part of the wave are equivalent (see part 3 where there are equal parts sine and cosine) and decreases the more the values differ (which corresponds to the angle between the real and imaginary parts in the plan getting smaller). A 80/20 split is larger than a 60/40 split, which is equal to a 40/60 split, which is less than a 20/80 split (equal to the 80/20 split).

Part C: Rect Function

1. Compute the Fourier Transform of the Rectangular Pulse (*rect*) and plot the magnitude and Power Spectrum:

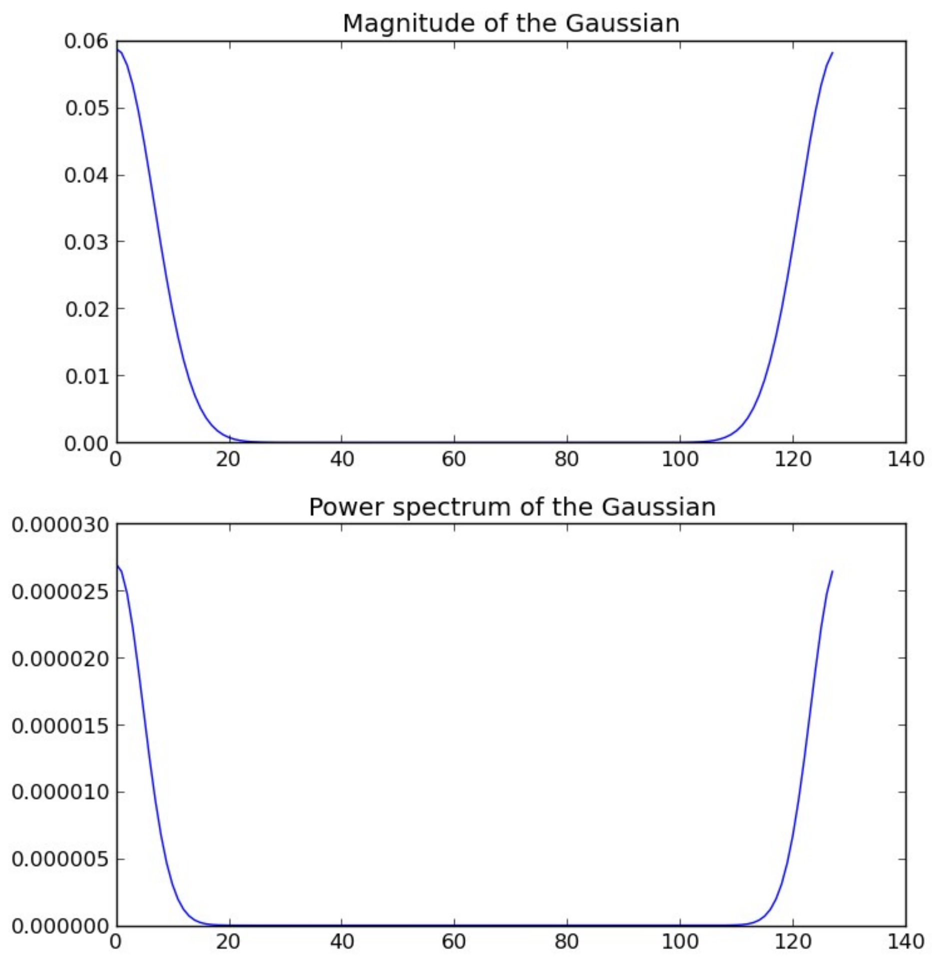


2. Explain why I get this result:
The rectangular pulse is like a square pulse that sits at 1 between the appropriate *a* values as described in the slides. Because of this, its magnitude is similar to what we expect of square-like pulse, where the Fourier transform is a sinc function starting high at the origin and bouncing lower as the frequency goes from 0 out. The magnitude, like seen in the previous parts, will have the same shape as the Fourier transform. We expect the Power spectrum to have the same shape as well because it is just a normalized version of magnitude and thus doesn't change anything. As you can see in the image below of the original rectangular pulse, its K value is approximately 38, and since A = 1 and M = 128, AK/M = 38/128 = 0.296, which is about the value found on around both sides of the origin when you wrap the frequency domain signal around correctly (I'm not sure why the value immediately to the right of the origin is higher than this)

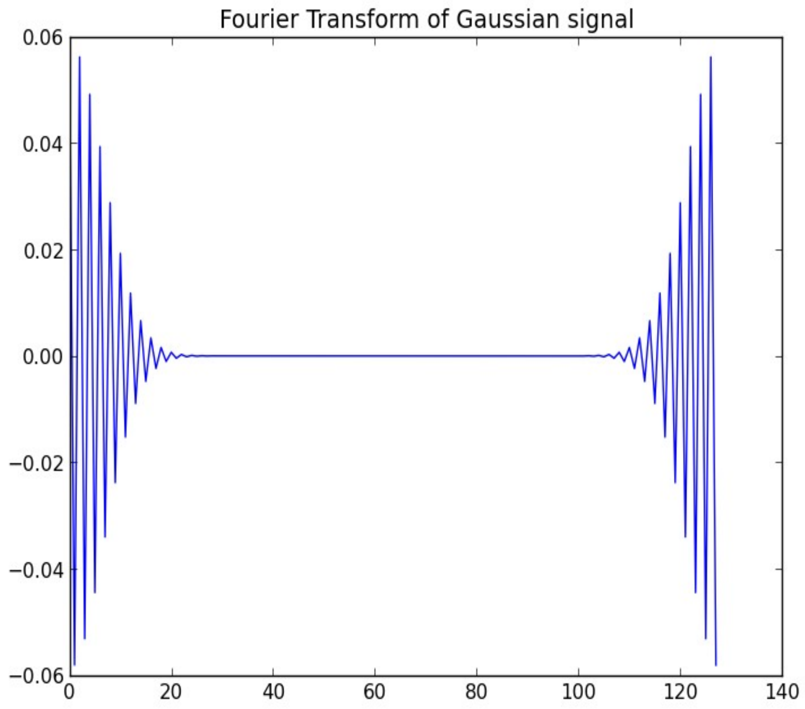


Part D: Gaussian Function

1. Compute the Fourier Transform of the Gaussian (*rect*) and plot the magnitude and Power Spectrum:



2. Explain (mathematically) why I get this result:
Since the Fourier transform of a Gaussian (as seen in the figure below) is equal to the Gaussian's signal representation in the time domain (e^(-pi*t^2) === e^(-pi*u^2)), it is expected that the graphs are similar as well. The magnitude also has a Gaussian/bell-curve shape.



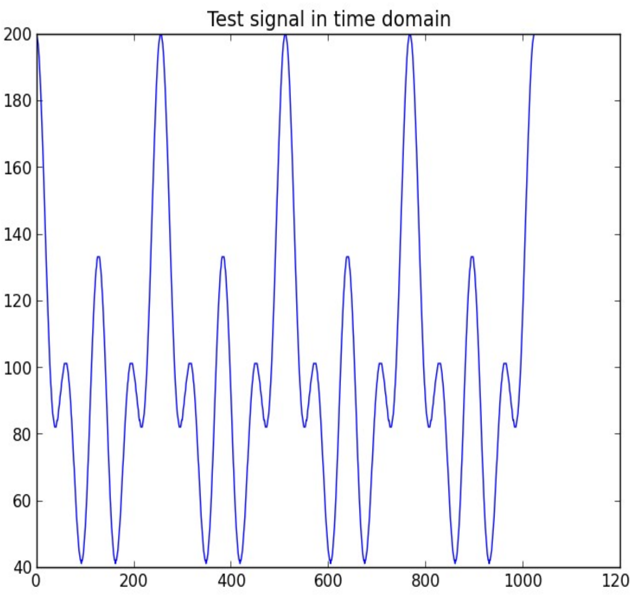
Part E: Frequency Analysis

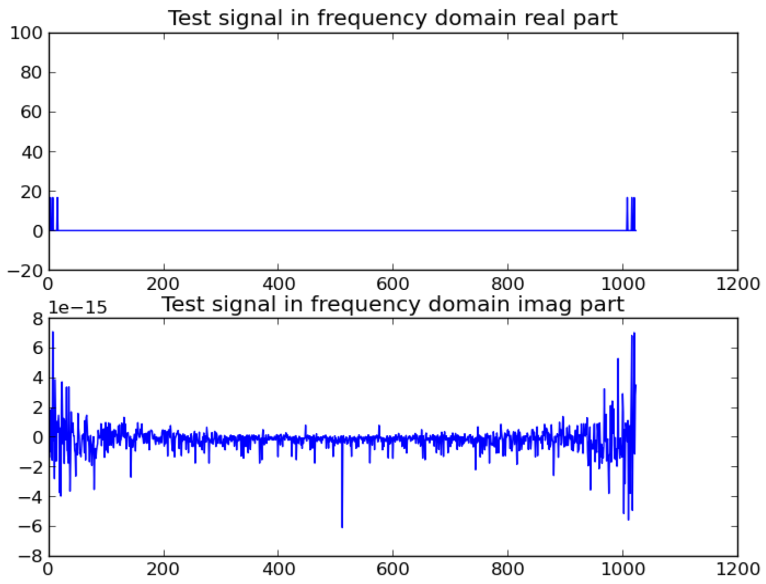
1. Program that uses the Fourier Transform to identify the *n* most significant frequencies in a signal.

```
def most_sig_freq(signal, n):
    fourier = dft_enhanced(signal)
    fourier = fourier[0: fourier.size/2]    # to not get the repititions
    listfourier = list(fourier)
    listfourier.sort(cmp=cmp_cplx, reverse=True) # gets lower to higher
    most_sig = list()
    for i in range(n):
        item = listfourier[i]
        most_sig.append(item)
    return most_sig
```

2. Three most significant frequencies in the 1-D signal found in (*test*) and plot of this signal in the time domain:

99.99609375+0j,
16.690694327617528+7.0637939941775585e-15j,
16.688324404331343+2.8796409701214998e-16j


Test signal in time domain

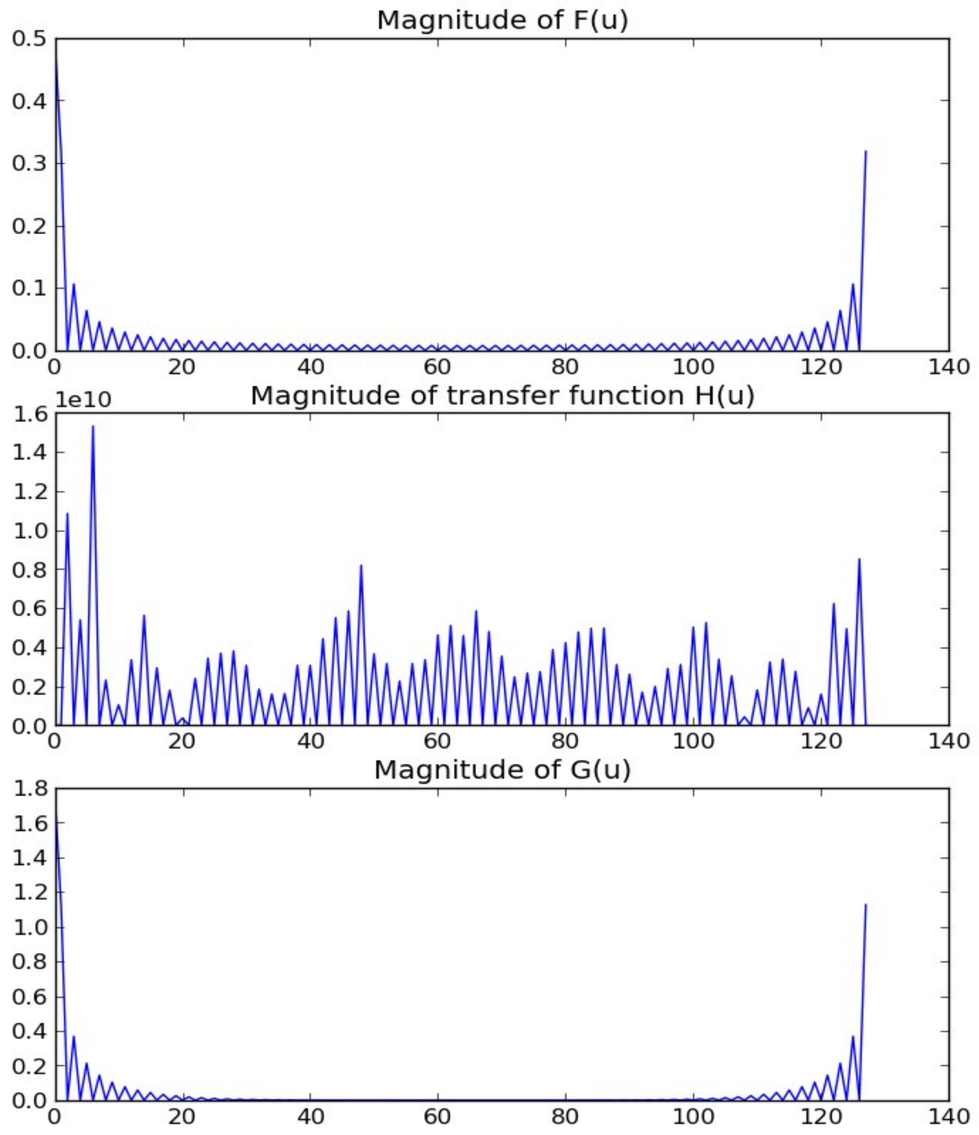

Test signal in frequency domain real part
Test signal in frequency domain imag part
1e−15

Part F: Discover the Transfer Function H(u)

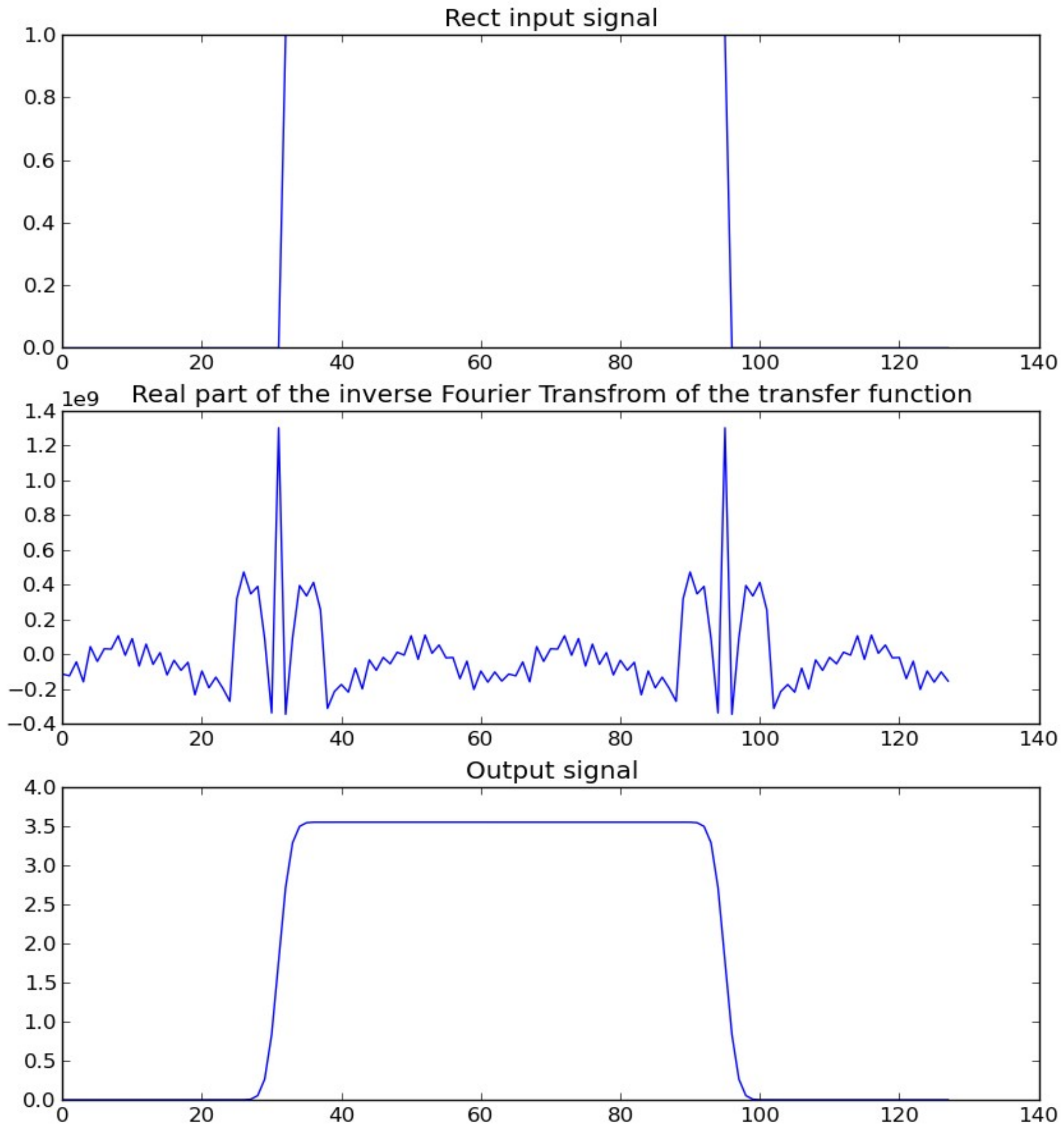
Let input f(t) = Rectangular Pulse and output g(t) = the output

1. Find the transfer function H(u) of the system used to produce this output and plot the magnitude portion of it.

*I interpreted this to mean "find the transfer function H(u), which transforms F(u) into G(u). That is, G(u) = F(u)*H(u), so H(u) = G(u)/F(u)"*



2. Apply the inverse Fourier Transform to the transfer function and plot the real part of the results. Does this bear any relationship to the input and output signals?
It seems the relationship between the input and output signals and the real part of the transfer function is visible in the large spikes around the value 31 and 95 in the middle figure below. It shows that the function uses a large amount of the input signal at those times to produce the output signal at those times (and immediately around this spike to produce the curve found in the output signal).



Notes: The most difficult thing I encountered was trying to understand the Fourier transform and specifically what each part of the wave represented with respect to the original input signal. The written portion took around an hour to complete. The coding part of the programming aspect part was relatively simple and straightforward. It took around 1.5 hours to program and 5 hours to do the reading, listening to Fourier transform lecture videos, speaking with the professor, the thinking, and finally the writing of the explanatory parts of the programming part.

Code:
```
import numpy as np
import matplotlib.pyplot as pl
import pickle as pk

# PART A: Implementt the Discrete Fourier Transform
# comparing this against numpy's np.fft.fft, they don't normalized to 1/M like we
do
def discrete_fourier_transform(signal_arr):
    siglen = len(signal_arr)    # M
    Fu = np.zeros(siglen, dtype=np.complex)
    for u in range(Fu.size):
        Fu[u] = np.complex(0, 0)
        for x in range(Fu.size):
            Fu[u] += np.complex(signal_arr[x] * np.cos(-2 * np.pi * u * x /
siglen), \
                        signal_arr[x] * np.sin(-2 * np.pi * u * x / siglen))
        Fu[u] /= siglen
    return Fu


# same as function above, just semi-enhanced/optimized
def dft_enhanced(signal_arr):
    siglen = len(signal_arr)
    # tabulate values of cos(2pik/M) and sin(2pik/M)
    cosarr = list(np.cos(2 * np.pi * k / siglen) for k in range(siglen))
    sinarr = list(np.sin(2 * np.pi * k / siglen) for k in range(siglen))

    Fu = np.zeros(siglen, dtype=np.complex)
    for u in range(Fu.size):
        Fu[u] = np.complex(0, 0)
        for x in range(Fu.size):
            k = (u * x) % siglen
            Fu[u] += np.complex(signal_arr[x] * cosarr[k], \
                        signal_arr[x] * -sinarr[k])
        Fu[u] /= siglen
    return Fu


def cmp_cplx(x, y):
    mx = np.square(x.real) + np.square(x.imag)
    my = np.square(y.real) + np.square(y.imag)
    if mx < my:
        return -1
    elif mx == my:
        return 0
    else:
        return 1


def most_sig_freq(signal, n):
    fourier = dft_enhanced(signal)
    fourier = fourier[0: fourier.size/2]    # to not get the repititions
    listfourier = list(fourier)
    listfourier.sort(cmp=cmp_cplx, reverse=True) # gets lower to higher
    most_sig = list()
    for i in range(n):
        item = listfourier[i]
        most_sig.append(item)
    return most_sig


# PART B: Simple sines and cosines
def partb():
    #B.1:
    #sine: f[t] = sin(2*pi*s*t/N) where s=8 and N=128
    #cos: f[t] = cos(2*pi*s*t/N) where s = 8 and N=128
    N = 128
    s = 8
    ftsine = np.zeros(N)
    ftcos = np.zeros(N)
    ftsincossum = np.zeros(N)
    for t in range(N):
        ftsine[t] = np.sin(2 * np.pi * s * t / N)
        ftcos[t] = np.cos(2 * np.pi * s * t / N)
    ftsincossum = np.add(ftsine, ftcos)

    allfts = [(ftsine, 'sine'),
              (ftcos, 'cosine'),
              (ftsincossum, 'sine + cosine')]
    #Graph problems 1 through 3
    for (ft, name) in allfts:
        #display the waves
        pl.plot(ft)
        pl.title('Original ' + name + ' wave')
        pl.show()

        #apply the dft to this sinusoid and plot the Real, Imaginary,
        #Magnitude, and Phase parts of the result
        fortran = dft_enhanced(ft)

        pl.subplot(2, 2, 1)
        pl.title('Real Part')
        pl.plot(fortran.real)
        pl.subplot(2, 2, 2)
        pl.title('Imag Part')
        pl.plot(fortran.imag)
        pl.subplot(2, 2, 3)
        mag = list(np.sqrt(np.square(n.real) + np.square(n.imag)) for n in
fortran)
        pl.title('Magnitude')
        pl.plot(mag)
        pl.subplot(2, 2, 4)
        fortranclean = map(clean, fortran)
        phase = list(np.angle(n) for n in fortranclean) #could also do
np.arctan2(n.real, n.imag)
        pl.title('Phase')
        pl.plot(phase)
        pl.show()

    # Graph problem 4 (playing with the relative weightings of the sine and cosine
parts)
    weightings = [(0.2, 0.8), (0.4, 0.6), (0.6, 0.4), (0.8, 0.2)]
    for (cosw, sinw) in weightings:
        #display the waves
        ftsincossum = np.add(sinw*ftsine, cosw*ftcos)
        pl.plot(ftsincossum)
        pl.title(str(sinw) + '*sine plus ' + str(cosw) + '*cosine wave')
        pl.show()

        #apply the dft to this sinusoid and plot the Real, Imaginary,
        #Magnitude, and Phase parts of the result
        fortran = dft_enhanced(ftsincossum)

        pl.subplot(2, 2, 1)
        pl.title('Real Part')
        pl.plot(fortran.real)
        pl.subplot(2, 2, 2)
        pl.title('Imag Part')
        pl.plot(fortran.imag)
        pl.subplot(2, 2, 3)
        mag = list(np.sqrt(np.square(n.real) + np.square(n.imag)) for n in
fortran)
        pl.title('Magnitude')
        pl.plot(mag)
        pl.subplot(2, 2, 4)
        fortranclean = map(clean, fortran)
        phase = list(np.angle(n) for n in fortranclean)
        pl.title('Phase')
        pl.plot(phase)
        pl.show()


def clean(x):
    newreal, newimag = x.real, x.imag
    if np.abs(x.real) < 0.001:
        newreal = 0
    if np.abs(x.imag) < 0.001:
        newimag = 0j
    newcomplex = np.complex(newreal, newimag)
    return newcomplex


def partc(signals):
    rectpuls = signals['rect']
    forrect = dft_enhanced(rectpuls)
    pl.subplot(2, 1, 1)
    pl.title('Magnitude of Rectangular Pulse')
    pl.plot(np.abs(forrect))

    #The Power Spectrum is the square of magnitude (so the sum of squares) of the
Fourier coefficients
    #but normalized by the number of samples.  It's the fraction of the
    #signal that's at a given frequency.
    pl.subplot(2, 1, 2)
    pl.title('Power spectrum of Rectangular Pulse')
    pl.plot(np.square(np.abs(forrect)) / forrect.size)

    pl.show()


def partd(signals):
    gaussian = signals['gaussian']
    forgaus = dft_enhanced(gaussian)
    pl.subplot(2, 1, 1)
    pl.title('Magnitude of the Gaussian')
    pl.plot(np.abs(forgaus))
    #Power Spectrum
    pl.subplot(2, 1, 2)
    pl.title('Power spectrum of the Gaussian')
    pl.plot(np.square(np.abs(forgaus)) / forgaus.size)

    pl.show()


def parte(signals):
    test = signals['test']
    sig = most_sig_freq(test, 3)
    fortest = dft_enhanced(test)
    print 'Most significant frequencies of \'test\' signal: %s' % str(sig)
    pl.title('Test signal in time domain')
    pl.plot(test)
    pl.show()
    pl.subplot(2, 1, 1)
    pl.title('Test signal in frequency domain real part')
    pl.plot(fortest.real)
    pl.subplot(2, 1, 2)
    pl.title('Test signal in frequency domain imag part')
    pl.plot(fortest.imag)
    pl.show()


def partf(signals):
    #I assumed we could use a built-in inverse dft here
    inputsf = signals['rect']
    outputsg = signals['output']

    Fu = dft_enhanced(inputsf)
    Gu = dft_enhanced(outputsg)
    #Find the transfer function H(u), which transforms F(u) into G(u)
    #That is, G(u) = F(u)*H(u), so H(u) = G(u)/F(u)
    Hu = np.divide(Gu, Fu) #complex division
    #1
    pl.subplot(3, 1, 1)
    pl.title('Magnitude of F(u)')
    pl.plot(np.abs(Fu))
    pl.subplot(3, 1, 2)
    pl.title('Magnitude of transfer function H(u)')
    pl.plot(np.abs(Hu))
    pl.subplot(3, 1, 3)
    pl.title('Magnitude of G(u)')
    pl.plot(np.abs(Gu))
    pl.show()

    #2
    pl.subplot(3, 1, 1)
    pl.title('Rect input signal')
    pl.plot(inputsf)
    ht = np.fft.ifft(Hu)
    pl.subplot(3, 1, 2)
    pl.title('Real part of the inverse Fourier Transfrom of the transfer
function')
    pl.plot(ht)
    pl.subplot(3, 1, 3)
    pl.title('Output signal')
    pl.plot(outputsg)
    pl.show()


def main():
    partb()
    signals = pk.load(open('HW4_signals.pkl', 'r'))
    # partc(signals)
    # partd(signals)
    # parte(signals)
    # partf(signals)


if __name__ == "__main__":
    main()
```