EECS 3451
Laboratory 1

PYTHON INTRODUCTION

In this lab, you will be introduced to the Python language. We will use the Jupyter Notebook environment to allow you to interact with the Python interpreter and to easily document your work. The Notebook environment works within your web browser and provides the "front end" or user interface. You will enter and evaluate expressions containing complex numbers. Expressions are entered using the Python language syntax. A special interactive version of Python - `IPython` - is the "back-end" which takes your expressions, works on them and provides the output to the Notebook for rendering within the browser application.

Python's plotting capabilities will be introduced and used extensively. You will also use the Notebook to create and run some simple signal processing functions that are used in later laboratory assignments. Note that this set of Lab instructions has been produced in a Notebook.

OBJECTIVES

By the end of this laboratory assignment, you should be able to:

1. Use Python to perform complex arithmetic.
2. Generate and plot signals and complex valued functions.
3. Confidently develop Python functions and save/document results of computations in the Notebook.

REFERENCE

Review Topics:

1. Algebra of complex numbers
2. Sketching of discrete and continuous time signals
3. Vector and matrix algebra

LAB ACTIVITIES

As you work through the problems and experiments in this course, you will be learning the Python language - an open source, high-level, object-oriented programming language. Python includes libraries of functions (modules) tailored for the needs of signal processing, communication, and control tasks. It is gaining acceptance by professionals in industry and academia worldwide in research, development, and design.

The first of these modules - `numpy` (Numerical Python) - works on matrices of numbers. We focus mostly on one-dimensional matrices called vectors that contain signal samples, or on multiple-dimensional matrices containing several signals or the parameters of a system. For example, a vector could contain just a list of values from a mathematical function that you wish to plot. We will first focus on familiarizing you with the matrix notation in IPython, and get you used to working with vectors and matrices in arithmetic operations.

A second module - `matplotlib` (Plotting) - provides the ability to create plots both within the Notebook environment and in separate windows.

This lab has three parts: reading while doing, doing guided by knowing expected results and assigned problems. Since the main difficulties in learning Python are in learning the syntax and (in some cases) learning to program, we hope that this format will remove some of the trauma. Your solutions to the questions in the “Quiz” section are be handed in as your Lab 1 report.

This introduction is not intended to present you with everything you need to know about Python; it is merely to bring you to a point where you can do the other labs in the course. Use the on-line help, and references for additional information.

You should follow this text with the Notebook environment running, and work through the examples and questions. Python is available on PCs, and many other platforms and runs under Windows, Linux, or Mac OS.

All versions of Python are compatible in file storage format and file format, so data stored on one system can be transferred to another without loss. Each Notebook session has one window which contains cells. A cell may be either:

1. A `Markdown` cell which contains headings, comments or explanations. They may also contain formatting using the Markdown language.
2. A `Code` cell which will contain Python code.

Part I Guided Activities

The following sections highlight some useful Python commands by working through some example problems. You should work along with the text.

Start the Notebook

This tutorial assumes that you have installed the Anaconda3 Python environment. This may be freely downloaded from the Continuum web site using this link:http://continuum.io/downloads#py34 (http://continuum.io/downloads#py34) *(Make sure you click on the "I want Python 3.4" link)*

Once installed, find the Anaconda Launcher icon on the desktop and double click. Select the *Launch* button for `ipython-notebook` application. After a minute or so a browser window will open up - this will provide a list of files in your home directory. It is recommended that you browse to a directory for the course and select (upper right drop down menu) `New -> (Notebooks) Python 3`. This will open up the Notebook with a blank first cell.

When the text window opens, a prompt appears:

```
In [ ]:
```

All code will be entered after a prompt like this one. To change the cell to a `Markdown` type, click on the drop down menu to the right of the `Code` type and change to the desired cell type.

To execute the current cell hit `Cntl-Enter`. To execute the current cell and advance to a new/next cell hit `Shift-Enter`.

Take a moment now to create a first level heading cell with the text "Lab 1 Introduction to Python" or similar. Also add a second level heading "Tutorial Part 1".

Evaluating Complex Variables and Expressions

In the following text, information that you enter will be in a Notebook Code cell

Problem 1: Express each of the following complex numbers in Catesian form, i.e., $s = a + jb$, where $a = Re(s)$ and $b = Im(s)$. Plot part (a) in the complex plane.

a. $je^{j11\pi/4}$

b. $(1-j)^{10}$

```
In [3]: import numpy as np
        j = np.complex(0,1)

        j*np.exp(11*j*np.pi/4)
```

```
Out[3]: (-0.70710678118654835-0.70710678118654668j)
```

Python initially does not have the `numpy` module loaded, so we load all of its functions using the `import` command. Note that we use the `np` prefix as a convention - to call any of the `numpy` functions we need `np.`.

The IPython environment allows tab auto-completion. As you type in `np.` hit the tab key and see a list of available `numpy` functions. Once you have selected the function move the cursor within the parentheses and hit `Shift-Tab` to see help on the function (also hit the '+' icon in the upper right to get extended help.

Note also that we must establish the variable `j` as the unit of imaginary numbers ($\sqrt{-1}$).

We then type in the expression for (a) using our variable `j`. Once done hit `Cntl-Enter` and the result is evaluated and placed in the `Out [ ]:` cell below the input Code cell.

Also, just like any programming language, exp (x) returns e^x. Note however that the function is contained within the `numpy` module, hence the `np` prefix. Other standard functions, including trigonometric functions, are available within `numpy`. Additionally, pi is defined as a special variable having the value π. Any special variable will act as defined until you change its value by assigning a new value to it. For example, the value of π from `numpy` is:

```
In [39]: np.pi
```

```
Out[39]: 3.141592653589793
```

For example, to change pi to 3, issue the command:

```
In [40]: #Skip this cell after kernel re-start...
         #np.pi = 3
         #np.pi

         #Another way to fix:
         #import scipy as sp
         #np.pi = sp.pi
```

To check that the value has been overwritten we can re-run the first cell to see a different result. Umm - OK but now π has an incorrect value. How do we recover? Simply by restarting the kernel - click on `Kernel -> Restart`. This forces the system to reset. Then just don't re-run the `np.pi = 3` cell again.

We can enter the equation for (b) in Problem 1 now:

```
In [41]: (1-j)**10
```

```
Out[41]: -32j
```

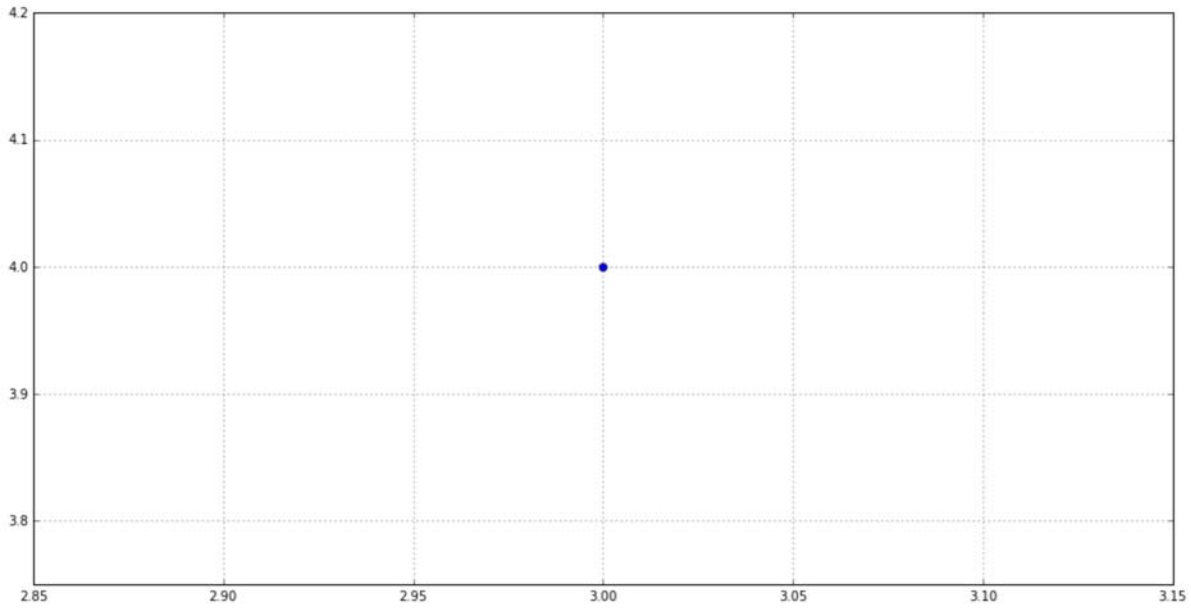
Plotting Complex-Valued Functions

Plots in Python are generated using the matplotlib module.

`plot (x, y)` generates a plot where the values of the vector x indicate points along the horizontal axis corresponding to the values in the vector y that are to be plotted on the vertical axis. Vectors x and y must have the same number of elements.

Since complex values have two components corresponding to $a + jb$, Python provides the real and imag functions to separate the real and imaginary parts of an imaginary number:

In [4]:
```
%matplotlib inline
import matplotlib.pyplot as plt

z = 3+4*j
zr = np.real(z)
zi = np.imag(z)
plt.figure(figsize=(16.0,8.0))
plt.plot(zr,zi,'o')
plt.grid()
plt.show()
```



Line that begin with % are called "magic". In this case the matplotlib magic tells the Notebook to produce the plot "inline" or in the Notebook itself. Another possibility here is "qt" (i.e. %matplotlib qt) to produce a separate plot window which allows you to zoom in and out and save in a different format. Take a moment to try this as well.

The ‘x’ parameter to the plot function tells Python to generate an x shape for each data point instead of a ‘connected-dot’ display. Since we only plotted one data point, this is extremely useful. In general you should always label axes on your plot and include a title. help plot shows you the other characters that can be used as well as the different colors that can be used on the plot.

Vectors and Matrices

Matrices and vectors make up the heart of numpy's capabilities. In this section, matrix and vector manipulations will be introduced. A vector is a one-dimensional set of values, an m x 1 or 1 x m matrix. In numpy this an **ndarray** data type. Vectors hold single signals or lists of data. They can be assigned a name and treated as any other variable in Python; however, operations performed on vectors are done element by element.

As an example of this, consider the function y = 3x+2. If we want to plot y as a function of x, we first create an x vector containing data points in the range of interest. Suppose the range is 0 to 5, using every integer point. There are several ways to create this data set with 'numpy. The first way is to type in every point:

```
In [43]: x = np.array([0, 1, 2, 3, 4, 5])
         type(x)
```

```
Out[43]: numpy.ndarray
```

This generates a row vector x, i.e. a 1 x 6 matrix containing six elements, the integers 0 through 5. Note that by simply entering the values, Python does not echo the results. An easier way to generate this same vector is to use a range-generating statement:

```
In [44]: x = np.arange(6)
         x
```

```
Out[44]: array([0, 1, 2, 3, 4, 5])
```

The arange function outputs an array with 6 elements with the default start being 0 and default step size of 1. A different step size, positive, negative, real or integer, can be specified by placing the step value between the beginning and end of the range, as in z below:

```
In [45]: z = np.arange(0,5,0.01)
         '...last 10 entries ',z[-10::], np.size(z)
```

```
Out[45]: ('...last 10 entries ',
          array([ 4.9 ,  4.91,  4.92,  4.93,  4.94,  4.95,  4.96,  4.97,  4.98,  4.99]),
          500)
```

generates 500 data points that are 0.01 apart, starting from 0 and ending at 4.99 (i.e. one step BEFORE the specified end of 5).

The next step is to evaluate the function y, using the x as defined above:

```
In [46]: y = 3*x+2
         y
```

```
Out[46]: array([ 2,  5,  8, 11, 14, 17])
```

This statement instructs numpy to multiply every element in x by 3 and then add 2 to every element, storing the results in y. Thus 3*x is treated as scalar multiplication of a vector and the 2 is implicitly treated as a vector of the same length as x comprising all 2s.

Since numpy is based on matrix operations, it is important to recall that you can only add or subtract matrices having the same dimensions, e.g., the addition of a 3x2 matrix with a 2x3 matrix is undefined. Matrix multiplication requires that the number of columns in the first matrix be the same as the number of rows in the second matrix. For example, multiplication of a 2x5 matrix A with a 5x3 matrix B results in a 2x3 matrix $C = AB$, whereas the multiplication BA is undefined. However, the multiplication $D = B^T A$ is defined, where ^{T} denotes the transpose operation.

Generating Complex Functions

Let's generate values for the complex function $f(t) = 3e^{j3\pi t}$ for *t* ranging from 0 to 1 in 0.001 increments. The first step is to create a time variable; note the use of the : operator with a noninteger step size.

```
In [6]: t = np.arange(0,1.001,0.001)
```

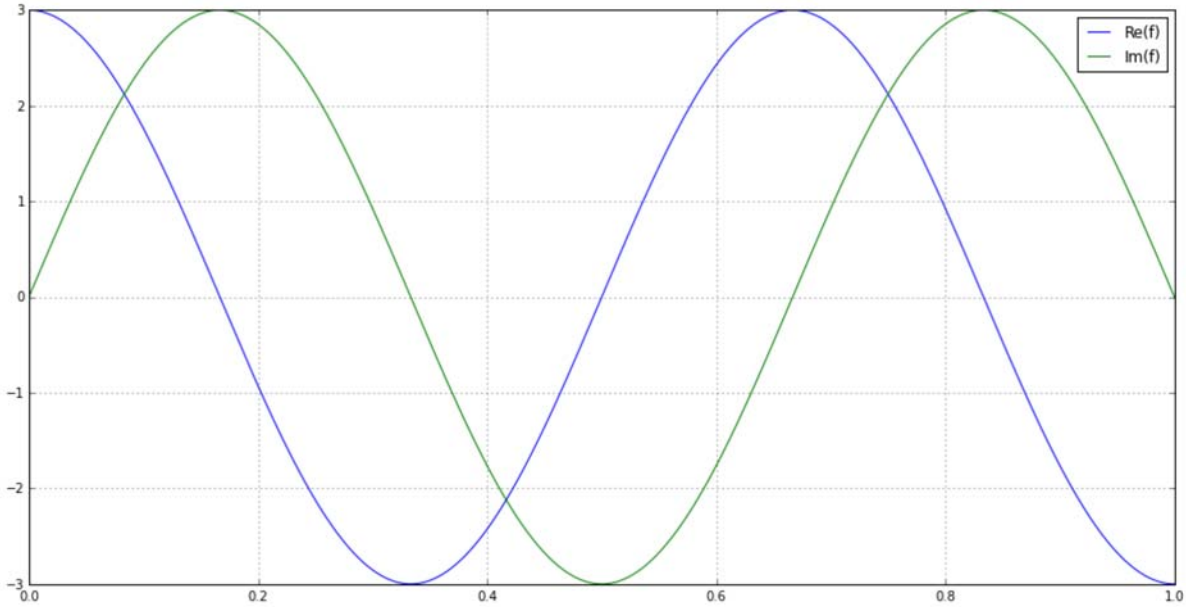
Here we simply type the command and do not ask for *t* to be displayed, unless you really want to see and wait for all 1000 values to be echoed back on your screen.

Next, construct a vector containing values of this function for each time value in *t*:

```
In [7]: f = 3*np.exp(j*3*np.pi*t)
```

It should be pointed out that transcendental functions (e.g. sin, cos, exp) in numpy work on a point-by-point basis; in the above command, the function *exp* computes a vector where each element is the exponential of its corresponding element in $j3\pi t$ (1001 total elements).

In [8]:
```
plt.figure(figsize=(16.0,8.0))
plt.plot(t,np.real(f), label='Re(f)')
plt.plot(t,np.imag(f), label='Im(f)')
plt.grid()
plt.legend()
plt.show()
```



Accessing Vectors and Matrices

The data in vectors can be viewed and displayed in several different ways: it can be plotted, printed to the screen, printed on paper, and saved electronically. It is not, however, always desirable to access the entire vector at once when displaying the information in it. To access single elements or ranges of a vector, an index element or list that identifies which elements are of interest is needed.

Elements in `numpy` vectors are identified by the vector name and an integer number or index, much in the same way that discrete time (DT) signals are indexed by integer values. However, in `numpy`, only positive integer indices are used. Thus the first element in a row or column vector f is denoted by $f(0)$, the second element by $f(1)$, and so forth. To access specific elements in a vector, you need to use the name of the variable and the integer index numbers of the elements vector you wish to access. Range statements can be used for indices to access the indexed elements much in the same way that range statements are used to define vectors comprising values in a specified range. For example,

In [50]:
```
v1 = f[24]
v2 = f[2:9]
v3 = f[0:50:2]
```

The first line accesses the 25th element of f. The second accesses elements 3 through 10, inclusive; and the third statement returns the odd-numbered elements between 1 and 50 (third value indicates the index increment between sliced values).

The second and third examples above are "slices" of the array f in that they extract multiple values.

Elements in matrices require use of two-dimensional indices for identification and access. For example, $A[2,1]$ returns the element in the third row, second column of matrix A; ranges can also be used for any index. For example, $A[0:3,3:8]$ defines a matrix that is equivalent to a section of the matrix A containing the first, second, and third rows, and the fourth through eighth columns.

If a ":" is used by itself, it refers to the entire range of that index. For example, a 3 x 5 matrix could have its fifth column referenced by $A[:,4]$, which means “all rows, 5th column only,” as well as $A[0:3,4]$, which means rows 1 to 3, 5th column only.

The index number can be another variable as well. This is useful for creating programming loops that execute the same operations on the elements of a matrix.

From other computer programming experiences, you should be familiar with the idea of creating a loop to repeat the same task for different values. Here’s an example of how to do this using Python. Suppose we want to generate an output vector where each element is the sum of the current element in the input vector and the element from 10 back in the input vector. The task to be repeated is the sum of two elements; we need to repeat this for each element in the vector past 10. The elements of the vector x define the input ramp function to be integrated, y will hold the result, and k is the loop index:

```
In [51]: x = np.array([3*k+2 for k in np.arange(0,5.1,0.1)])
         y = np.array([x[k-10]+x[k] for k in np.arange(11,len(x),1)])
         y

Out[51]: array([  7.6,   8.2,   8.8,   9.4,  10. ,  10.6,  11.2,  11.8,  12.4,
                 13. ,  13.6,  14.2,  14.8,  15.4,  16. ,  16.6,  17.2,  17.8,
                 18.4,  19. ,  19.6,  20.2,  20.8,  21.4,  22. ,  22.6,  23.2,
                 23.8,  24.4,  25. ,  25.6,  26.2,  26.8,  27.4,  28. ,  28.6,
                 29.2,  29.8,  30.4,  31. ])
```

We accomplish this task in two lines which use two separate *list comprehensions*. A list is a Python data type similar to a `numpy` array but a bit more general. In the above example we want the output as a `numpy` array (hence the np.array() function is used). We pass in a list (which is always enclosed in square brackets []). The list is created via hte list comprehension which is a sort of implicit loop. We define how each value in the list is calculated (the statement inside the loop) and then the range of values used to create the list entries (the loop index).

The first line forms the input vector x (as a `numpy` ndarray) using the variable k which itself is a list of values from 0 to 5 in steps of 0.1.

In the second line, the variable k (re-used) is set to range from 11 to the length of x; this allows k to index all elements in x. k is set to increment by integers.

Note that Python can use explicit loops, but these are generally avoided in order to make the code more readable. For speed reasons rewrite your operations in terms of vector additions and multiplications instead of looping.

For example, we can rewrite this problem to use vector addition by creating two new vectors, one which is x offset by 10 and the other which is x padded with 10 zeros (since we can only add vectors of similar lengths).

In [52]:
```
x1 = np.concatenate((np.zeros(10),x), axis=0)
x2 = np.concatenate((x,np.zeros(10)), axis=0)
y = x1+x2
y = y[11:len(y)-10]
y
```

Out[52]:
```
array([  7.6,   8.2,   8.8,   9.4,  10. ,  10.6,  11.2,  11.8,  12.4,
        13. ,  13.6,  14.2,  14.8,  15.4,  16. ,  16.6,  17.2,  17.8,
        18.4,  19. ,  19.6,  20.2,  20.8,  21.4,  22. ,  22.6,  23.2,
        23.8,  24.4,  25. ,  25.6,  26.2,  26.8,  27.4,  28. ,  28.6,
        29.2,  29.8,  30.4,  31. ])
```

The above may seem like more operations, however using vector addition in Python executes faster than using for loops. To prove this we can use Python's profiling tools to test the speed of our code.

Suppose we set up a Python function as follows:

In [53]:
```
def method1(x):
    return np.array([x[k-10]+x[k] for k in np.arange(11,len(x),1)])
```

We have defined a Python function `method1` which is passed an array x and returns an array using a list comprehension (as we did above).

We now define a competing function `method2`:

In [54]:
```
def method2(x):
    x1 = np.concatenate((np.zeros(10),x), axis=0)
    x2 = np.concatenate((x,np.zeros(10)), axis=0)
    y = x1+x2
    return y[11:len(y)-10]
```

To test which method is faster we use the `timeit` magic within the Notebook. For `method1` we have:

In [55]:
```
%%timeit
method1(x)
```

```
10000 loops, best of 3: 109 µs per loop
```

and for `method2` we have:

In [56]:
```
%%timeit
method2(x)
```

```
10000 loops, best of 3: 26.8 µs per loop
```

which seems to suggest that the vector addition method is faster (by a factor of ~3.5). Sometime the message received may indicate something about cacheing - running the cell again may remove this warning.

Storing Results and ipynb Files

Usually you want to save the results that you have generated during a Notebook session, including data vectors created and commands used to process them. This can be accomplished by:

1. Simply saving the Notebook in its native .ipynb format. The name of the file may be changed at any time by clicking on the field to the right of "jupyter" at the top of the Notebook window.
2. Saving the file to "static" format such as pdf. Select `File -> Download as -> HTML (.html)` You can then simply print to pdf format by opening the html file in your browser and printing to pdf format.

It is recommended that you become familiar with ipynb files. They are extremely useful and will save you much time and effort.

Part II: Guided Activities with Answers

Now that you have been introduced to Python and Notebooks, larger, more complex and exciting problems await! This section will present an example of how to solve a problem using the Notebook. In addition, you will be given several problems with answers, but not the steps taken to generate the answers. You should work through these problems during the laboratory session.

Question 1

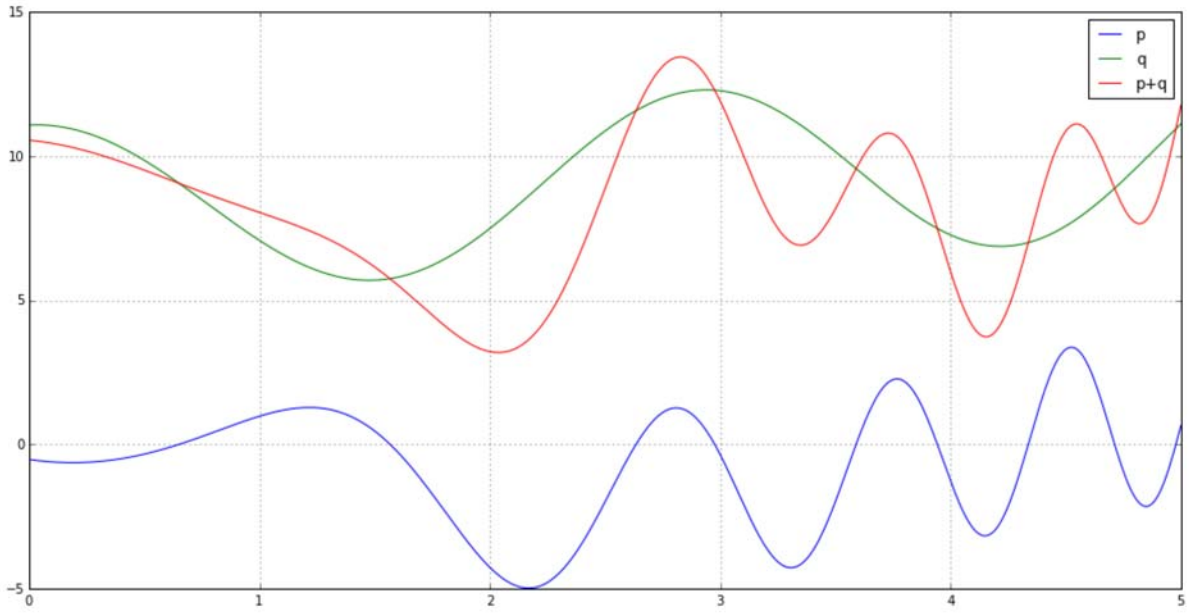
Add the functions $p = 3\sin(x^2) + 2\cos(y^3)$ and $q = 3\cos(xy) + 2y^2$ for x in the range 0 to 5 with the added constraint that: $y = 0.05x + 2.01$

Use increments of 0.01. Plot all three functions.

Solution to Question 1

This can be done in two different ways. The easiest way is to make two vectors containing the individual functions and a third vector containing the sum. The alternative is to algebraically substitute in x, y, p and q into the total expression and reduce. This is extremely unpleasant.

In [57]:
```
#Comments in the code can be placed like this

# establish the x vector
x = np.arange(0,5.01,0.01)
# ..and the y vector
y = 0.05*x + 2.01

# Now for the p vector
p = 3*np.sin(x*x)+ 2*np.cos(y*y*y)
plt.figure(figsize=(16.0,8.0))
plt.plot(x,p, c = 'blue', label = 'p')

#Next q
q = 3*np.cos(x*y) + 2*y*y
plt.plot(x,q, c='green', label = 'q')

#...and finally p+q
r = p + q


plt.plot(x,r,c = 'red', label = 'p+q')
plt.legend()
plt.grid()
plt.show()
```

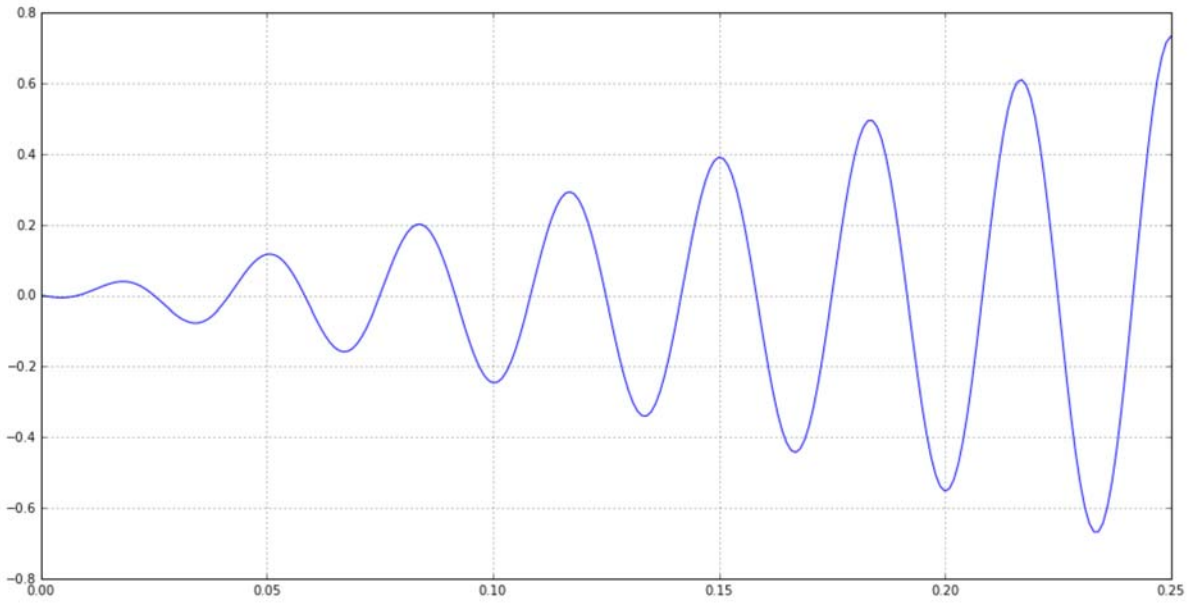


Question 2

Plot the function $y(t) = (1 - e^{2.2t})\cos(60\pi t)$. Use t from 0 to 0.25 in 0.001 increments.

Solution to Question 2

In [72]:
```
#Enter your own code here...
```



Question 3

A polynomial function has roots at -2, 2, $-2+3j$, $-2-3j$. Determine the polynomial, plot the four roots in the complex plane, and plot the polynomial function for the range $-5 \leq x \leq 5$ in steps of 0.01. You may wish to use `Help -> NumPy` to look up the functions *polyfromroots* and *polyval* which are part of the `numpy.polynomial` module. To import this (sub) module use an import like:

In [59]:
```
from numpy.polynomial import polynomial as poly
```

Solution to Question 3

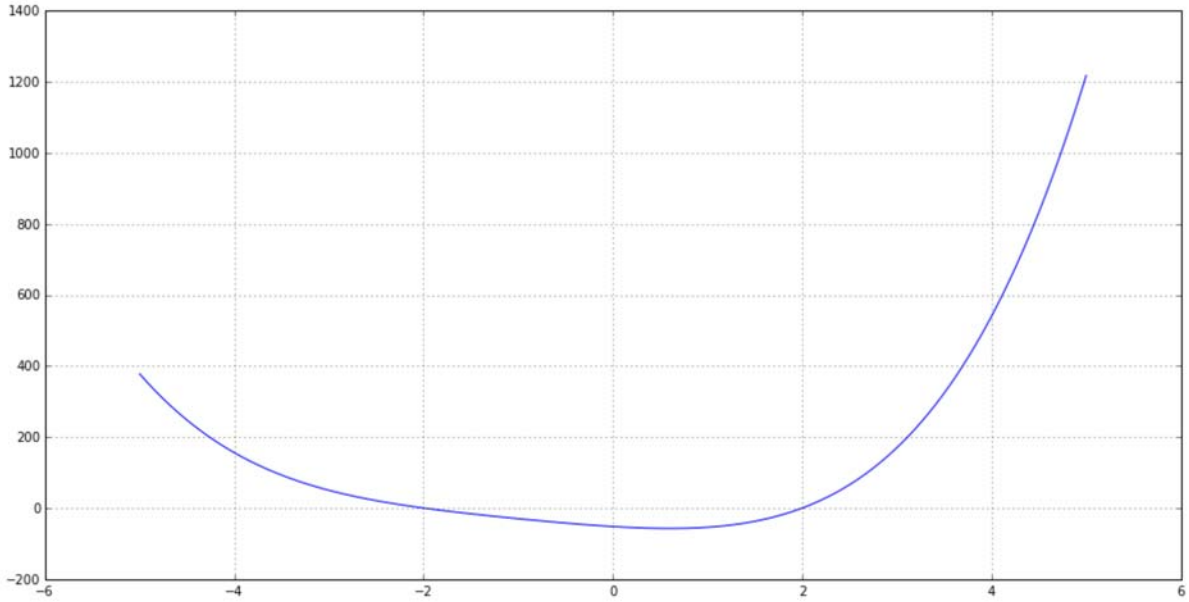
Coefficients:

In [60]:
```
#Enter your own code here...
```

```
Polynomical coefficients [-52.+0.j -16.+0.j   9.+0.j   4.+0.j   1.+0.j]
```

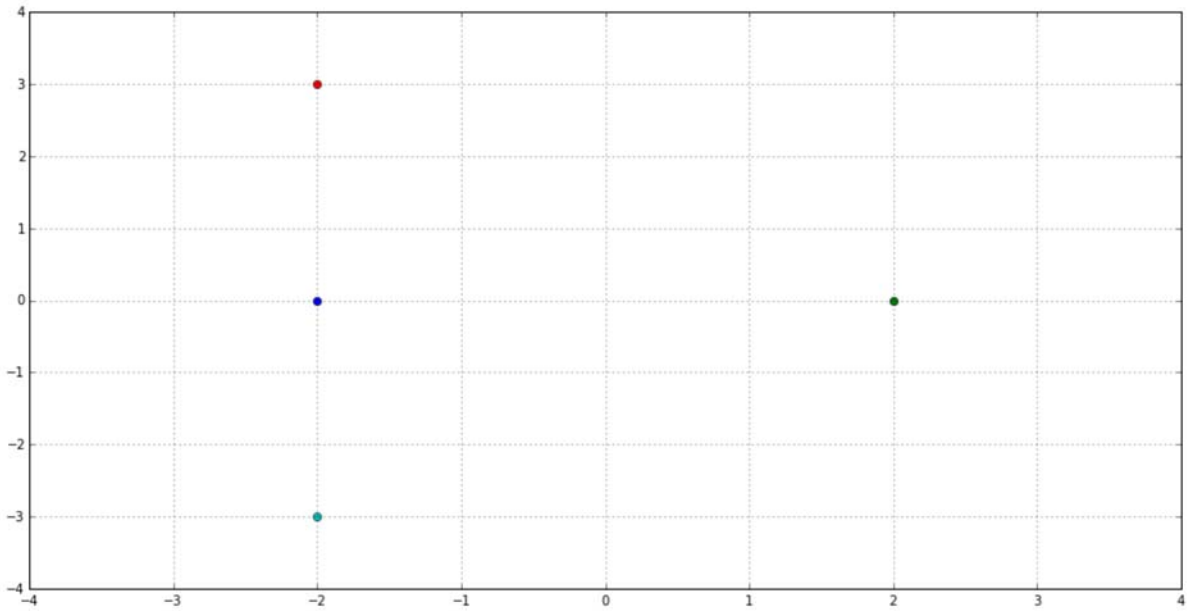
Plot of funcion:

In [61]:
```
#Enter your own code here...
```



Plot of polynomial roots in complex plane:

In [62]:
```
#Enter your own code here...
```



Question 4

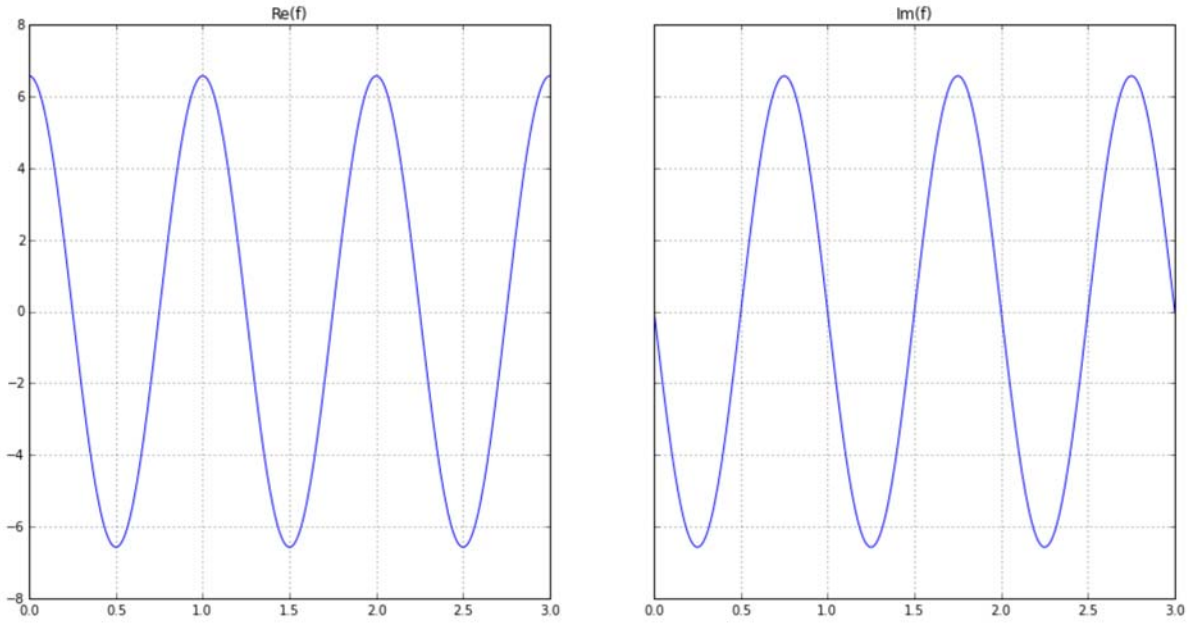
The complex function $f(t)$ has the form: $f(t) = 3e^{-j2\pi t+\pi/4}$

Plot the real and imaginary parts as a function of time from 0 to 3 seconds in 0.01-second increments. Also plot the magnitude and phase of f as a function of time. You may wish to look up the functions subplot, title, and plot to see how to generate more than one plot in the graphics window at the same time. You will also need abs and angle. These commands are very useful in signals and systems.

Solution to Question 4

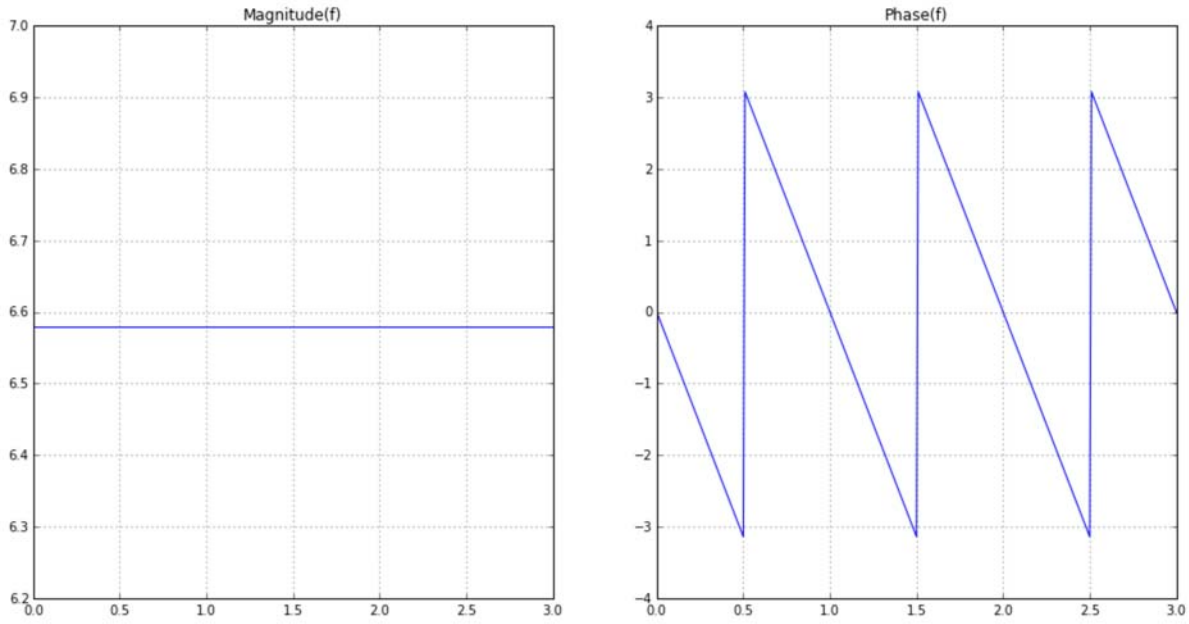
Plot of Real and Imaginary components of $f(t)$

```
In [63]: #Enter your own code here...
```



Plot of Magnitide and Phase of $f(t)$

In [64]:
```
#Enter your own code here...
```



Creation of a Python Module

A module in Python is file containing functions (and/or classes). The file name is the name of the module with a `.py` extension. Once you have created a module you can import it and use its functions just as would use say a `numpy` function.

A useful feature of functions is that the lines that follow the function definition may begin with a doc string symbol (''') are printed when help is requested for your function (`Shift-Tab` is pressed within the function parentheses).

Make a module function that takes two variables as arguments, adds 1 to the first variable, multiplies the second variable by two, and returns the product of the two variables. This function is called `blackbox` and the module will be called `bbx.py`.

The module might look something like this:

In [65]:
```
%%writefile bbx.py
# %load bbx.py

def blackbox(a, b):
    """
    Blackbox( a, b)

    Returns (a+1)*2*b

    where a and b are numpy 1d arrays
    If a and b are not the same length,
    the larger variable is stripped to be
    the same size as the smaller
    @author: H. Chesser
    """

    #Determine the length ot each vector
    la, lb = len(a), len(b)

    #Add 1 to a
    a = a+1

    #Multiply b by 2
    b = b*2

    if la < lb:
        return a*b[:la]
    else:
        return a[:lb]*b
```

```
Overwriting bbx.py
```

The elements in the above file that are required in order for Python to recognize it as a valid function are (1) the module filename must have a .py extension, (2) the function must start with the word def to indicate to Python that it is a function that may require inputs, (3) and, finally, there may be returned data (an array in this case although this is not always necessary for a valid function).

Note also that once we type in the code we can write it out to a file using the `%%writefile` magic or if its existing we can load it with the `%load` magic.

To use the module we must first import it. Then we are able to call any of its functions as shown below:

In [66]:
```
import bbx
# Note we could have said "import bbx as bbx" but its redundant

#function inputs a, b
a = np.array([1, 2, 3])
b = np.array([1, -5, 10, 2])

#call the function blackbox from module bbx:
bbx.blackbox(a, b)
#Try placing the cursor between the function parentheses and hit Shift-Tab to see t
he doc string
```

Out[66]:
```
array([  4, -30,  80])
```

Question 5

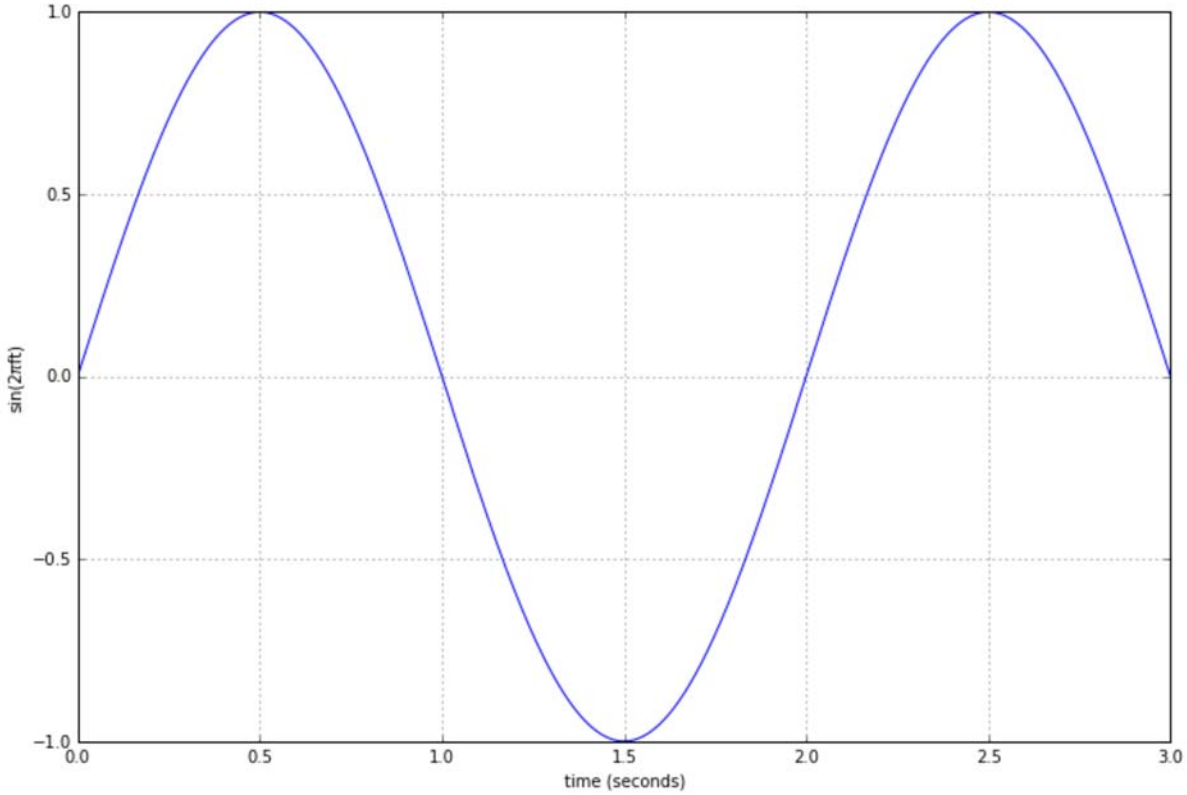
Generate a Python module that calculates a sine wave of f Hz (f is the function input) for 3 seconds using 0.001-second increments and plots the sine wave versus time. Display the length of the time sample and the length of the sine wave calculated. Bonus if you can get axes labelled as shown.

Solution to Question 5

In [67]:
```
%%writefile myfuncs.py
# Enter your own code here...
```

```
Overwriting myfuncs.py
```

In [68]:
```
#Example of running the code
# Labels sometimes disappear - restart kernel
import myfuncs as my


samples = my.sine(0.5)

print("Samples in the time series =",samples)
```




```
Samples in the time series = [  0.00000000e+00   3.14158749e-03   6.28314397e-03
 ...,   6.28314397e-03
   3.14158749e-03   3.67394040e-16]
```

Question 6

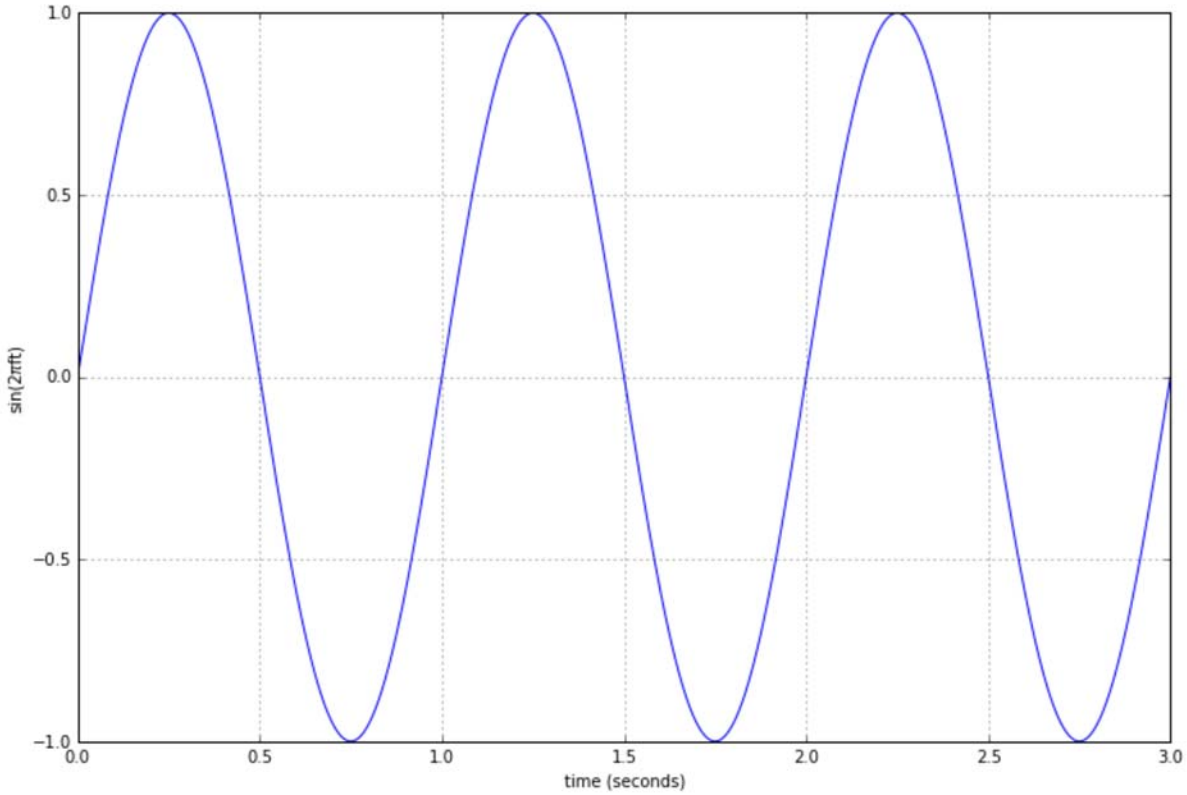
Add a function to the module you developed in Question 5 called `squarer` that returns the input vector with each element squared. Modify your sine function to return the sine array that you plotted. Write a final function in the same module that calculates and returns the sum of the elements of an input array. Use this module then to calculate the sum of both the sine wave vector and the vector composing square of the sine wave. (Hint: Save your module, now with three functions into a different module name - ex- `Question6.py`).

Solution to Question 6

In [74]:
```
%%writefile myfuncs.py
# %load myfuncs.py
# Enter your own code here...
```

```
Overwriting myfuncs.py
```

In [70]:
```
#Example use of new functions
import myfunctions as my

y = my.sine(1.0)

ysq = my.squarer(y)

y_sum = my.summer(y)

ysq_sum = my.summer(ysq)

print("Sum for sine wave =", y_sum)
print("Sum for sine wave squared =", ysq_sum)
```




```
Sum for sine wave = -5.68434188608e-14
Sum for sine wave squared = 1500.0
```

Note that the sum of a sine squared should be 0.5 if we interpret the sum as an approximation to an integral. The function returns 1500 because there are 1000 samples per period and three periods. 1500 / 3000 is 0.5. You must be cautious when interpreting results because you are dealing with discrete elements in the vector instead of a true continuous function.

Note also that if the sum of the sine function accurately approximates the integral, it should be 0. It is, instead, a very small number. This error is due to rounding errors in the least significant place of every sine value calculated and is an artifact of numerical computing. If you expect a sum to be equal to zero, don't be surprised if it is very small instead of zero.

ASSIGNED PROBLEMS (PART III)

Submit solutions to the following problems with your lab write-up.

Problem 1

Compute:

$$\frac{(2+j5)(1-j5)}{(1+j7)(3+j2)}$$

and express your answer in both rectangular and polar coordinates.

Problem 2

An interesting plot is generated by the complex function $r = 1 - \cos\theta$, where r is the radius of a complex number z expressed in polar coordinates and θ is the angle. Sweep θ from 0 to 2π and plot the real and imaginary parts of $z = re^{j\theta}$ as x and y coordinates, respectively.

Problem 3

In communications, a signal is called a power signal if it has a zero time average

$$\bar{x} = \frac{1}{N}\sum_{k=0}^{N-1} x[k]$$

and a nonzero, finite time average of its square

$$\bar{x^2} = \frac{1}{N}\sum_{k=0}^{N-1} x^2[k]$$

In contrast, a message signal has a nonzero time average and a nonzero, finite time average of its square.

Create a function that determines if a signal is a message signal or a power signal. It should return the value 1 if the signal is a message, and 0 if the signal is power. Test your function with the power signal $\sin 10t$ and the message signal $rand(t)$. Let t = [0,1] and use increments in time of 0.01. Use the `numpy rand` function to get a normal (i.e., Gaussian) distribution (see the on-line help for `rand`).

Problem 4

Some Python functions require that a vector representing time samples be in the first column of a matrix and the corresponding signal values at each time be in the second column. Such a matrix, called A, has been created. Write down three Python expressions to extract the time vector into the variable t, the signal vector into the variable x, and the 100th time/signal pair into the matrix y. Hint: Use slicing.

Problem 5

Write a Python function `half` that removes every other element from an arbitrary length vector, creating a shorter vector made of only the odd-numbered elements of the original vector; and a Python function `double` that creates a longer vector by adding an additional element between neighboring elements in the original vector. Each new element should equal the average of its neighboring elements. Use only matrix/ vector manipulations; do NOT use loops. Test your solution by applying `half` and then `double` to the vector x = [1, 2, ... 6, 7, 6, ... 2, 1]. What happens after the execution of `half` followed by `double`, and `double` followed by `half`?