122nd ASEE Annual Conference & Exposition
June 14 - 17, 2015
Seattle, WA

Paper ID #12461

# A Taste of Python - Discrete and Fast Fourier Transforms

**Dr. Mohammad Rafiq Muqri, DeVry University, Pomona**
**Mr. Eric John Wilson**

Electrical and electronics engineering student

**Dr. Javad Shakib, DeVry University, Pomona**



©American Society for Engineering Education, 2015

# A Taste of Python - Discrete and Fast Fourier Transforms

This paper is an attempt to present the development and application of a practical teaching module introducing Python programming techniques to electronics, computer, and bioengineering students at an undergraduate level before they encounter digital signal processing and its applications in junior or senior level courses.

The Fourier transform takes a signal in time domain, switches it into the frequency domain, and vice versa. Fourier Transforms are extensively used in engineering and science in a vast and wide variety of fields including concentrations in acoustics, digital signal processing, image processing, geophysical processing, wavelet theory, and optics and astronomy. The Discrete Fourier Transform (DFT) is an essential digital signal processing tool that is highly desirable if the integral form of the Fourier Transform cannot be expressed as a mathematical equation. The key to spectral analysis is to choose a window length that suits the signal to be analyzed, since the length of the window used for DFT calculations has a substantial impact on the information the DFT can provide. The operation count of the DFT algorithm is time-intensive, and as such a number of Fast Fourier Transform methods have been developed to adequately perform DFT efficiently.

This paper will explain how this learning and teaching module was instrumental in progressive learning for students by presenting Python programming and the general theory of the Fourier Transform in order to demonstrate how the DFT and FFT algorithms are derived and computed through leverage of the Python data structures. This paper thereby serves as an innovative way to expose technology students to this difficult topic and gives them a fresh taste of Python programming while having fun learning the Discrete and Fast Fourier Transforms.

## 1. Background

Engineering departments are often confronted with the necessity to update laboratory exercises and equipment with the latest emerging technological trends within tight budget constraints. Another challenge faced by departments pertains to satisfying the Engineering Technology Accreditation Commission (ETAC) criteria for capstone senior project experience within the curriculum. In this paper we will explain how we attempted to solve these challenges by exposing students to new emerging software trends for computational applications in engineering and life sciences.

## 2. Introduction

Although DSP algorithms may be implemented on any powerful processor, specialized digital signal processing hardware enables the greatest speed and efficiency.[4] Digital signal processors designed specifically for operations common in DSP have special features that permit them to accomplish in real time what other processors cannot achieve. Real time means that outputs keep pace with the collection of input samples during actual operations. For some operations such as filtering, this means a new output sample can be produced as each new input sample is received. For others such as FFTs, the output information can be produced only when a block of input samples has been recorded.   A DFT decomposes a sequence of values into components of

different frequencies. This operation is useful in many fields, but computing it directly from the definition is often too slow to be practical.

A FFT is a way to compute the same result more quickly; computing a DFT of N points in the naive way, using the definition, takes $O(N^2)$ arithmetical operations, while an FFT can compute the same result in only O(N log N) operations. The difference in speed can be substantial, especially for long data sets where N may be in the thousands or millions. In practice, the computation time can be reduced by several orders of magnitude in such cases, and the improvement is roughly proportional to N / log (N). This huge improvement made many DFT-based algorithms practical; FFTs are of great importance to a wide variety of applications as described earlier.

## 3. Procedure

All engineering technology, computer information systems (CIS) and Networking students at our school take an introductory computing course where they use the Raspberry PI computer and an interactive shell named Integrated Development Environment (IDLE) for very basic Python programming. Python is available for download from https://www.Python.org/download which is also preinstalled on the Raspberry PI SD card.

Python is a general-purpose interpreted, interactive, object-oriented and high-level programming language. Python was created by Guido van Rossum in the late eighties and early nineties. It is an open source programming language that runs on many platforms including Linux, Mac OS X and Windows. It is widely used and actively developed, has a vast array of code libraries and development tools, and integrates well with many other programming languages, frameworks and musical applications.

## 4. Python for Scientific Computing

Python's scientific computing prowess comes largely from the combination of three related extension modules: NumPy, SciPy and Matplotlib. NumPy[5] [Oliphant, 2006] adds a homogenous, multidimensional array object to Python. It also provides functions that perform efficient calculations based on array data.

- NumPy, which stands for Numerical Python is written in C, and can be extended easily via its own C-API. As many existing scientific computing libraries are written in Fortran, NumPy comes with a tool called f2py which can parse Fortran files and create a Python extension module that contains all the subroutines and functions in those files as callable Python methods. With NumPy you will receive programs such as PI or the ability to easily create matrices.

- SciPy, which stands for Scientific Python, builds on top of NumPy, providing modules that are dedicated to common issues in scientific computing, and so it can be compared to MATLAB toolboxes. The SciPy modules are written in a mixture of pure Python, C and Fortran, and are designed to operate efficiently on NumPy arrays. A complete list of

SciPy modules is available online at http://docs.scipy.org, but examples include file input/output (scipy.io) and signal processing (scipy.signal). Built into the library, or package, of SciPy is the FFT, or fast Fourier Transform. This is a key word within the package. Once added to the code, we can call this function and pass in ant wave, and it will give us the Fourier Transform. We can then import the plot package and plot the FFT. In just four or five lines of code, it doesn't only take the FTT, but it is plotted as well. Below is an example of how this can be done.

NumPy is one of the main tools used in Python to perform math. Basic Python will give us basic addition subtraction and so on; It is NumPy that will give you access to complex arrays and constants such as PI. The extent of NumPy used in this program was some line spacing functions and PI, but the functions that NumPy offers are virtually endless. NumPy can do different statistics operations, binary operations, logical functions, sorting and searching functions; And the list continues on. In short, NumPy is ultimately a powerful tool in Python.

SciPy, on the other hand has a completely different set of functions. Together, SciPy and NumPy can provide thousands of functions that can be used. With SciPy one can perform functions such as integration, linear algebra, signal processing and of course the Fourier transform.

- Matplotlib is a library of 2-dimensional plotting functions that provides the ability to quickly visualize data from NumPy arrays, and produce publication-ready figures in a variety of formats. It can be used interactively from the Python command prompt, providing similar functionality to MATLAB or GNU Plot [Williams et al., 2011]. It can also be used in Python scripts, web applications servers, or in combination with several GUI toolkits.

## 5. Example

Compared with NumPy there is a colossal list of things one can do with SciPy. The following listing is what we use SciPy for in this instance.

```
import numpy as np
from scipy.fftpack import fft
import matplotlib.pyplot as plt
N = 600
T = 1.0/800.0
x = np.linspace(0.0, N*T,N)
a = np.sin(50.0 * 2.0*np.pi*x)
b = 0.5*np.sin(80.0 * 2.0*np.pi*x)
y = a + b
yf = fft(y)
Y = 2.0/N * np.abs(yf[0:N/2])
X = np.linspace(0.0, 1.0/(2.0*T), N/2)
import matplotlib.pyplot as plt
plt.plot(x,y)
plt.grid()
plt.show()
```

When we import a package, the key word to use this package is the name of the package itself. When importing NumPy, we would need to use the keyword “numpy” every time you wanted to use this package. To shorten how much you actually type you can name the package when you import it. The import statement in the code snippet previously stated begins with the line “import numpy as np”. The ‘np’ can be any name and can be abbreviated. For example instead of writing “numpy.sin” to use sine function, we can simply write “np.sin”.

Highly efficient algorithms for computing the DFT were first developed in the 1960s. Collectively known as Fast Fourier Transforms (FFTs), they all rely upon the fact that the standard DFT involves redundant calculation**.** Suppose we have a signal with N sample values, where N is an integer power of 2**.** We first separate x[n] into two subsequences, each with N/2 samples. The first subsequence consists of even number points in x[n], and the second consists of odd number points - writing n = 2k, when n is even, and n = 2k + 1 when n is odd. We can thus express the original N-point DSP in terms of two N/2 point DFTs. Now we can take the decomposition further by breaking each N/2- point subsequence down into two shorter, N/4-point subsequences. The process can continue until, in the limit, we are left with a series of 2-point subsequences, both of which require a very simple 2-point DFT. A complete decomposition of this type brings light to one of the commonly used radix-2, decimation in time, FFT algorithms.[1]

The code previously stated as an example starts off by importing the necessary packages to run this program. We then set N to a value of 600 and T to a value of 1.800. These are just constants used to set an endpoint and spacing for the time domain on the graph. This is also used to set “x” when creating the initial signal. We will demonstrate by using two separate signals and adding them together to better see the FFT in action. These two signals are actually random signals. One of the signals is half the size of the other, henceforth there will not be one continuous sine wave throughout the entire time. Any single wave can be composed of any number of other harmonic components. This is the reason we take the FFT of a signal. When we see a signal in the time domain, we cannot tell how many different individual signals are present. When we switch it to the frequency domain, it is evident how many waves are truly present. After taking the FFT of the signal, we find its absolute value. We then create a new variable ‘x’ because the indices of two variables have to be the same when graphing in Python. We plot the graph of the signal using the Matplotlib package.

The advantage of using this built-in function is that it will give a student a better understanding of the FFT itself. This would be a good demonstration to have the students perform before delving too deeply into the computations to show what is actually going on and why we use the FFT. This method does not need any powerful algorithm or advanced knowledge of the Python programming. This can serve as a stepping stone to introduce students to the usefulness of programming as well as a basic understanding of the FFT. The student would then be ready to develop and enhance powerful and fast algorithms for other engineering and scientific applications.

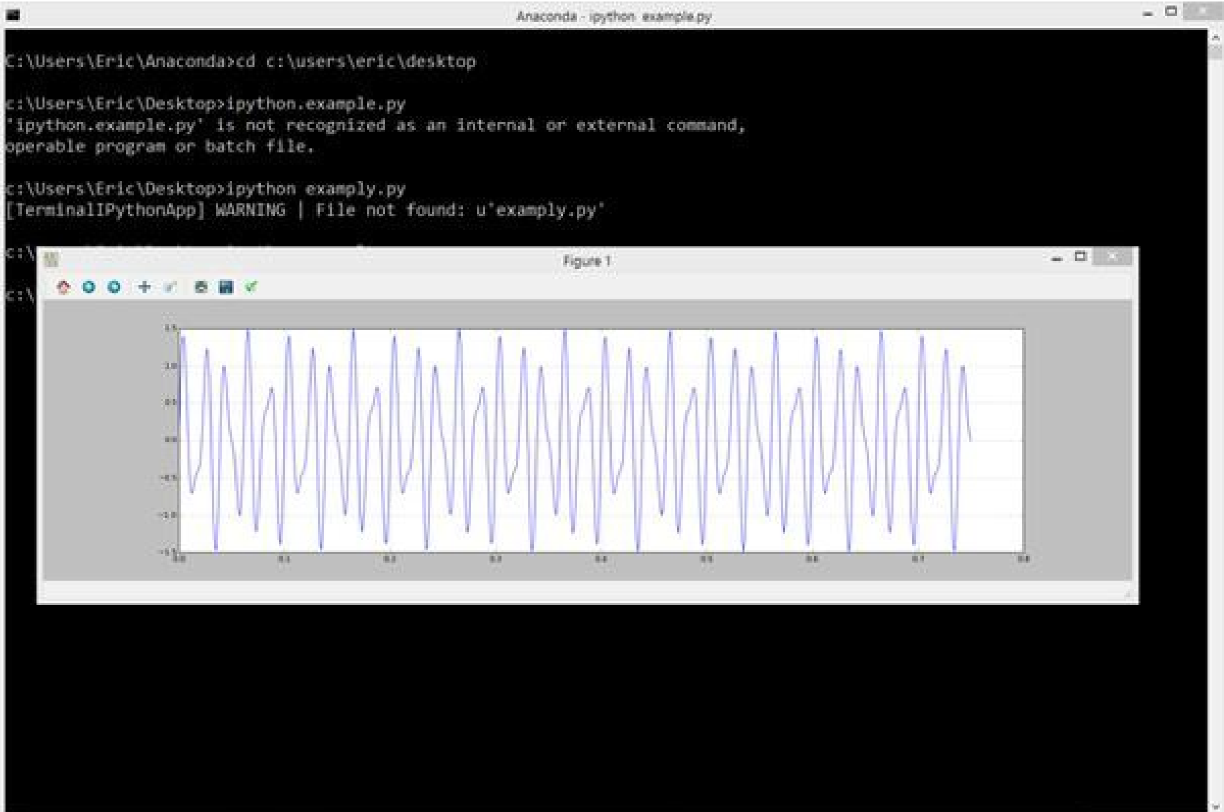


Figure 1. The plot of the signal in time domain.

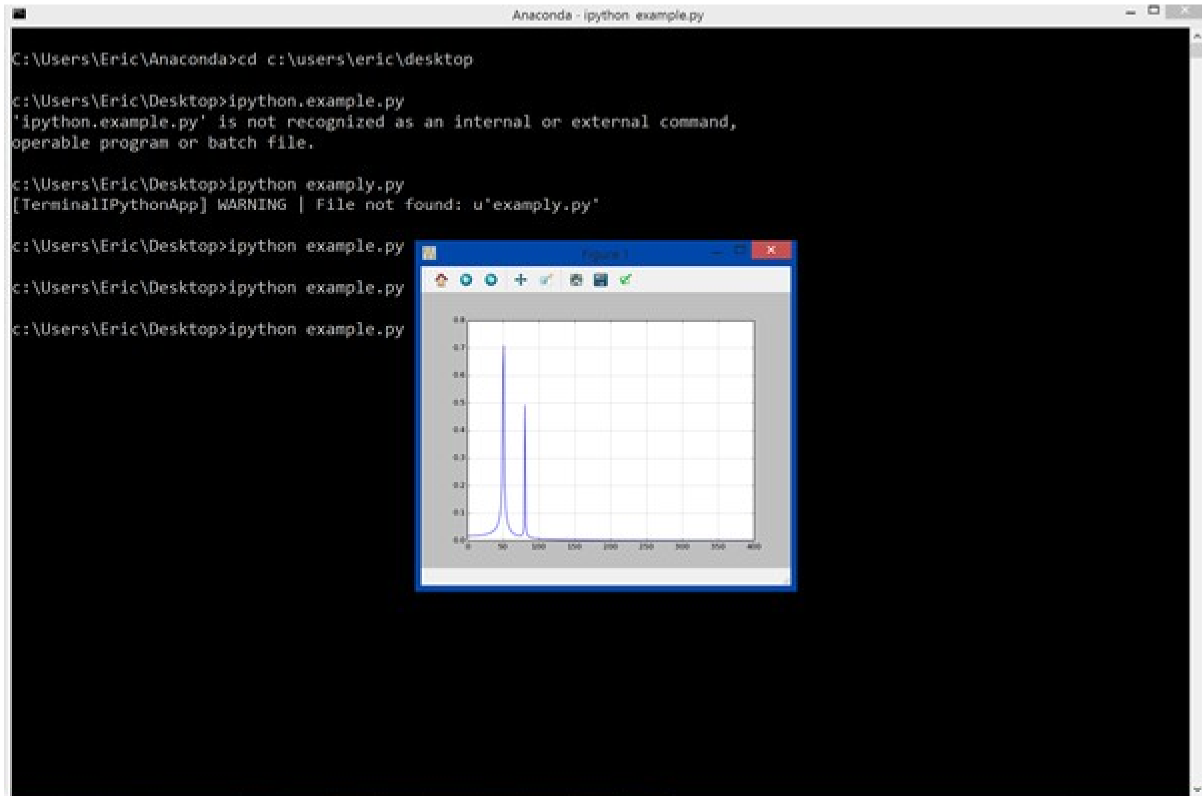


Figure 2. The FFT of the signal.

Plotting the function is the easiest part. Using numpy.linespace creates an array of numbers. This takes three parameters when initiated. The first is the starting number, the second is the spacing between the numbers and the third is the ending number. Here we go from 0.0 to N with a spacing of N*T. This will give us our traditional x-axis. Your y-axis will be the function that you want to graph; in this case we want to plot the FFT. When we call the FFT function, we set the value that it returns to the variable y. This variable has hundreds if not thousands of numbers that will make up the curve. We import the necessary library that is matplotlib.pyplot as plt and call the plot function and pass on x and y variables. The way pyplot works is that, once plotted, it will not automatically appear on the screen. We must call the function plt.show() in order to show the plot when the program is executed.

For novice programmers, this is the probably the easiest way to compute FFT using Python. We will now demonstrate another computing technique which is a bit tedious and requires coding, but one can delegate how the code will run.
Given below is an example of the instructor-led program which the student design enhanced, compiled and displayed the output. The intent of this program is to show how general Python workhorse discrete Fourier Transform and other control theory methods [5] can be introduced at the earliest stage to engineering technology students in conjunction with the tools and concepts that will further reinforce in future DSP courses.

The following listing is the code for computing FFT.

```
import numpy as np
import matplotlib.pyplot as plt
def discreteFT(fdata, N,fwd):
      X=[0]*(2*N)
      omega=0.0
      k, ki, kr, n = 0,0,0,0
      if(fwd):
            omega = 2.0*np.pi/N
      else:
            omega = -2.0*np.pi/n
      for k in range(0,N):
            kr = 2*k
            ki = 2*k+1
            X[kr]=0.0
            X[ki]=0.0
            for n in range(0,N):
                  X[kr] +=
fdata[2*n]*np.cos(omega*n*k)+fdata[2*n+1]*np.sin(omega*n*k)
                  X[ki] +=
fdata[2*n]*np.sin(omega*n*k)+fdata[2*n+1]*np.cos(omega*n*k)
      if(fwd):
            for k in range(0,N):
                  X[2*k]/=N
                  X[2*k+1] /= N
      return X

def fastFFT(fdata, N, fwd):
```

```
    omega, tempr, tempi, fscale= 0.0,0.0,0.0,0.0
    xtemp,cosine,sine,xr,xi=0.0,0.0,0.0,0.0,0.0
    i,j,k,n,m,M=0,0,0,0,0,0

    for i in range(0,N-1):
        if(i<j):
            tempr = fdata[2*i] tempi
            = fdata[2*i+1] fdata[2*i]
            = fdata[2*j]
            fdata[2*i+1]=fdata[2*j+1]
            fdata[2*j]=tempr
            fdata[2*j+1]=tempi
        k = N/2
        while(k <= j):
            j -=k
            k >>=1
        j+=k

    if(fwd):
        fscale = 1.0
    else:
        fscale = -1.0

    M = 2
    while(M<2*N):
        omega = fscale*2.0*np.pi/M
        sin = np.sin(omega)
        cos = np.cos(omega)-1.0
        xr = 1.0
        xi = 0.0
        for m in range(0,M-1,2):
            for i in range(m,2*N,M*2):
                j = i+m
                tempr = xr*fdata[j]-xi*fdata[j+1]
                tempi = xr*fdata[j+1]+xi*fdata[j]
                fdata[j]=fdata[i]-tempr
                fdata[j+1]=fdata[i+1]-tempi
                fdata[i]+=tempr
                fdata[i+1]+=tempi
            xtemp = xr
            xr = xr + xr*cos - xi*sin
            xi = xi+xtemp*sin+xi*cos
        M*=2
    if(fwd):
        for k in range(0,N):
            fdata[2*k]/=N
            fdata[2*k+1]/=N

def mainDFT():
    N = 64
    T = 2.0
    b = True
```

```
      fdata=[0]*(2*N)
      for i in range(0,N):
            fdata[2*i] = np.cos(4.0*np.pi*i*T/N)
            fdata[2*i+1] = 0.0
      Y=discreteFT(fdata,N,b)

      X = np.linspace(0.0,1.0/(2.0*T),N/2)
      yf = 2.0/N * np.abs(Y[0:N/2])

      plt.plot(X,yf)
      plt.grid()
      plt.show()


def mainFFT():
            N = 64
            T = 1.0
            tn,fk = 0.0,0.0
            fdata = [0]*(2*N)
                 for i in range(0,N):
                  fdata[2*i] =
np.cos(8.0*np.pi*i*T/N)+3*np.cos(14.0*np.pi*i*T/N)+
2*np.cos(32.0*np.pi*i*T/N)
                 fdata[2*i+1]=0.0

            y = discreteFT(fdata,N,True)
            fdata = (fdata[0:N])
            y = (y[0:N])
            x = np.linspace(0.0,1.0/(2.0*T),N)
            plt.plot(x,y)
            plt.grid()
            plt.show()
      mainFFT ()
```

The first output is shown in Figure 3. Because there are three signals that make up one signal, the wave looks a lot busier.

Next, we will show what this signal is like in the time domain to see each signal. In this example our signal comprises of three separate component waves so we get three spikes in our graph as seen in Figure 4.

There are two separate functions in this program: the DFT (Discrete Fourier Transform) and the FFT (Fast Fourier Transform). These are two different methods that essentially perform the same operation but use completely different algorithms to do so. The DFT algorithm is highly efficient and can perform computations at a much expeditiously than the traditional FFT can. This rate in speed is seen mostly when a very large amount of data, in the thousands or millions, that needs to be processed. With the small amount of data points shown in this program, there isn't a noticeable difference in the speed.

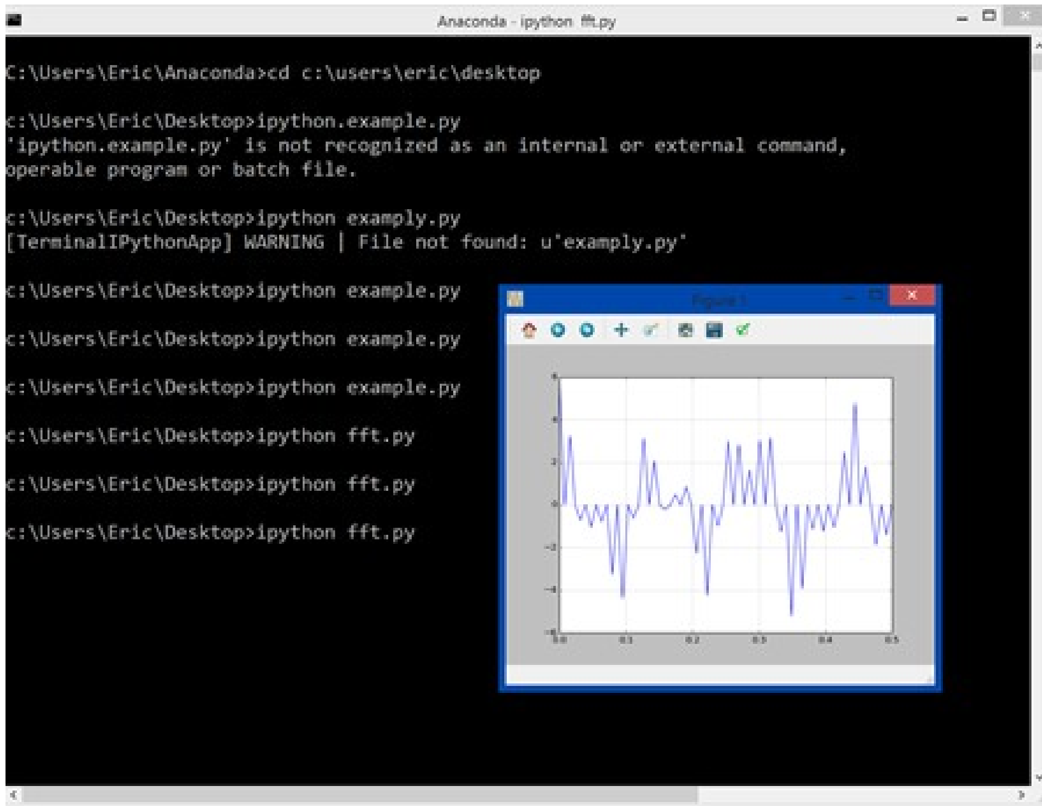


Figure 3. The signal before FFT taken in time domain

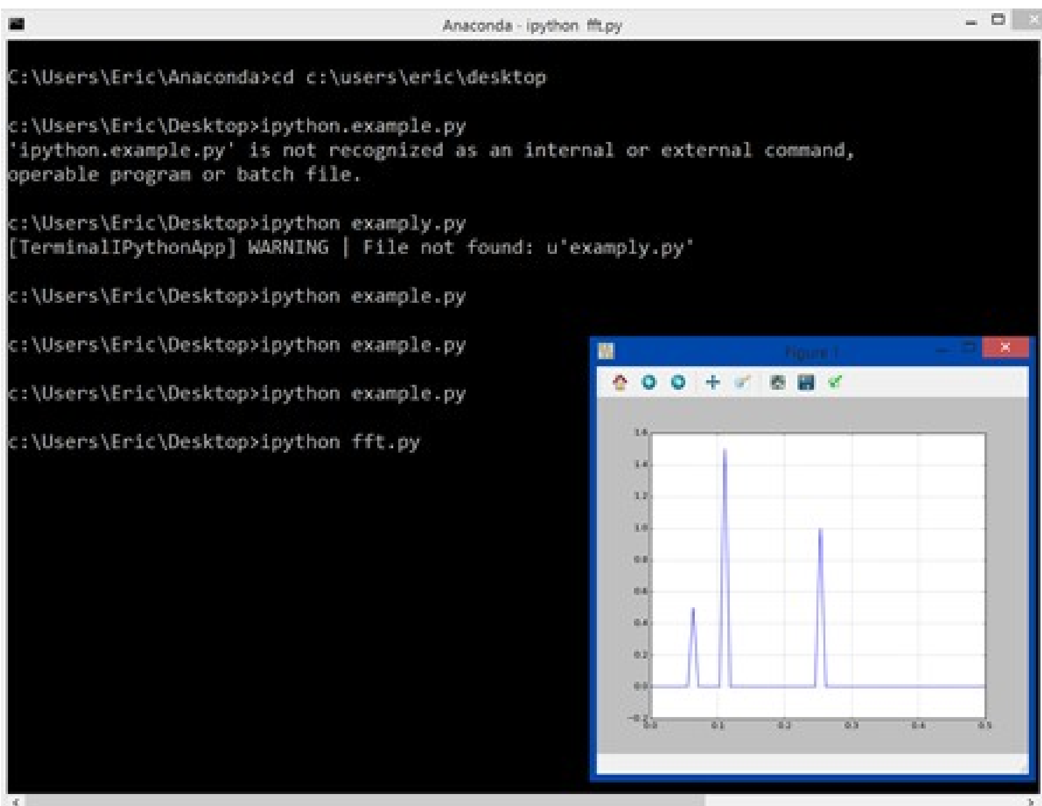


Figure 4. The FFT of the signal

## 6. Progressive Learning by Doing

Since programming with Python is straightforward, a useful supplement for this module is to have students implement some of the following applications after they are presented with the basics of FFT and DFT Python programming. The ability of students to transition into some of these numerous applications was an impetus for this paper.

- Network Socket Programming using Python
- Solving ODEs using Python
- Solving PDEs using Python
- Image Processing using Python

## 7. Results

To date, we have used this DFT, FFT Python education module for two different purposes. One was to teach basic Python programming to high school students; the other was to teach digital signal processing basics to burgeoning sophomores in the engineering technology program before they have even taken the DSP course in their junior year and work on their capstone senior project. In both cases, the delivery was well received and the students were able to understand most of the basic concepts within a very limited time.

## 8. Conclusions

Ultimately the hardware and software laboratory material developed in this paper was developed by students for students. With basic knowledge on how FFT's and DFT's can be computed as well as of the Python language; there should be no problem in writing these algorithms.
This paper presented a FFT, DFT teaching beginning module *that* is still evolving. It is expected to become a strategically integrated module *that* promotes visualization, intuitive explanation, and learning by doing pedagogical methods for future Python applications in scientific computing, nanotechnology, engineering and other emerging disciplines.

## Bibliography


1. Shakib, J., Muqri, M., *A Taste of Java: Discrete and Fast Fourier Transforms*, American Society for Engineering Education, AC 2011-451.
2. Lynn, Paul A., Fuerst, Wolfgang, *Introductory Digital Signal Processing with Computer Applications*, John Wiley & Sons, 1994.
3. R. Meyer, H.W. Schuessler, and K. Schwarz. (1990). FFT Implementation on DSP chips – Theory and Practice. *IEEE International Conference on Acoustics, Speech, and Signal Processing*.
4. Travis Oliphant. 2006. Guide to NumPy, Trelgol Publishing, USA.
5. H.V. Sorensen and C.S. Burrus. (1993, March). Efficient computation of the DFT with only a subset of input or output points. *IEEE Transactions on Signal Processing*, *41*(3), 1184-1200
6. Joyce Van de Vegte, *Fundamentals of Digital Signal Processing*, Prentice Hall, 2002.
7. Discrete Fourier Transform (numpy.fft) 2014, http://docs.scipy.org/doc/numpy/reference/routines.fft.html