Implementation of Discrete Fourier Transform and Orthogonal Discrete Wavelet Transform in Python

Tariq Javid Ali[1], Pervez Akhtar[2], Muhammad Faris[3], Idris Mala[4], Syed Saood Zia[5]

[1,2,3] *Hamdard Institute of Engineering and Technology, Hamdard University, Karachi-74800, Pakistan*
[4] *Computer Science Department, Usman Institute of Technology, Karachi-75300, Pakistan*
[5] *Computer Engineering Department, SSUET, Karachi-75300, Pakistan*

[1] tariq.javid@hamdard.edu.pk
[4] imala@uit.edu
[5] szia@ssuet.edu.pk

Abstract—This paper presents implementation of Discrete Fourier Transform and Orthogonal Discrete Wavelet Transform in Python computer programming language. The Fourier Transform is a fundamental signal processing tool whereas the Wavelet Transform is a powerful and advanced signal processing tool. Both have applications in numerous scientific and engineering disciplines. Our implementation aims to develop a deeper understanding of these transformations by presenting detailed coding steps to generate the frequency-domain and wavelet-domain outputs for selected example time-domain input signals. The results generated from developed program code are compared using built-in functions with similar matches have shown the successful implementation.

Keywords**—**DFT; DWT; FFT; Python; Convolution

I. INTRODUCTION

Python has emerged recently as a computer programming language of choice for science and engineering disciplines. Despite presence of famous powerful computer languages, for example C/C++/C# and Java, and mathematical tools, for example MATLAB and MAPLE, this computer programming language is making its way towards new heights [1-2]. The language is open source with an easily understandable syntax and is supported by a large community of programmers all around the world. Recently, many courses have replaced their adoption of computer language by Python, for example [3] replaced Java with Python as the Python code is easier for the novice learner. The major strengths of this programming language are modularity and ability to integrate with different computer programming languages [4].

Discrete Fourier Transform (DFT) is a fundamental signal processing tool. On the other hand, Discrete Wavelet Transform (DWT) is a powerful and advanced signal processing tool. Both tools have a wide range of applications in many scientific and engineering disciplines. These are implemented in almost all computer programming languages and mathematical software tools. Therefore, learning to use application of DFT and DWT on time-domain input signals to generate corresponding frequency-domain and wavelet-domain representations is an established exercise for students in science, technology, engineering, and mathematics (STEM) programs. Use of computer programs and mathematical software tools is a common practice to perform lengthy calculations.

The purpose of this study is to explore how to learn fundamental and advanced mathematical formulations, for example DFT and orthogonal DWT, by using a prospective computer programming language. The work in this paper aims to strengthen the understanding of DFT by implementing circular convolution and Fourier transformation and also to strengthen the understanding of DWT by implementing orthogonal Wavelet transformation in Python. A step-by-step approach is presented which is useful for readers even if they are unfamiliar with this computer programming language. In addition, examples are presented to use Numpy [5] built-in Fast Fourier Transform (FFT) function to compute the DFT and PyWavelets [6] built-in function to compute the DWT. The resulting spectrum and scalogram from selected example time-domain signals by using the developed Python program code are compared with outputs using built-in functions. Similar matches show a successful implementation of both DFT and DWT.

II. CONCEPTS AND MATHEMATICAL EXPRESSIONS

In this section, a review of related mathematical expressions with corresponding matrix views is presented from [7-8]. This section and most of the examples used in this study are selected from this very useful reference.

A. Discrete Fourier Transform

The circular convolution is closely related to DFT and for any two length-N sequences $x_n = \{x_0, x_1, \dots, x_{N-1}\}$ and $h_n = \{h_0, h_1, \dots, h_{N-1}\}$, it is defined as

$$(Hx)_n = \sum_{k=0}^{N-1} x_k h_{(n-k) \bmod N} = \sum_{k=0}^{N-1} h_k x_{(n-k) \bmod N} \qquad (1)$$

where H is called the circular convolution operator associated with hn. The result (Hx)n is also a length-N sequence. The related matrix view is given below.

$$Hx = \begin{bmatrix} h_0 & h_{N-1} & \dots & h_1 \\ h_1 & h_0 & \cdots & h_2 \\ \vdots & \vdots & \iddots & \\ h_{N-1} & h_{N-2} & \cdots & h_0 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \end{bmatrix} \qquad (2)$$

where H is a circulant matrix with hn as its first column. The DFT of a length-N sequence xn is defined as

$$X_k = (Fx)_k = \sum_{n=0}^{N-1} x_n W_N^{kn}, k \in \{0,1,\dots,N-1\} \qquad (3)$$

where X_k is called the spectrum of sequence x_n and $W_N^{kn} = e^{-j(2\pi/N)kn}$ is a unit-modulus Eigen sequence. "… the DFT arises from identifying the unit-modulus Eigen sequences of the circular convolution operator …" [7]. The related matrix view is given below.

$$Fx = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & W_N^2 & \cdots & W_N^{N-1} \\ \vdots & \vdots & \iddots & \vdots \\ 1 & W_N^{N-1} & \cdots & W_N^{(N-1)^2} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \end{bmatrix} \qquad (4)$$

The IDFT of a length-N sequence is defined as

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k W_N^{-kn}, n \in \{0,1,\dots,N-1\} \qquad (5)$$

where. $W_N^{-kn} = e^{j(2\pi/N)kn}$ The related matrix view of IDFT is given below.

$$x_n = \frac{1}{N} \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & W_N^{-2} & \cdots & W_N^{-(N-1)} \\ \vdots & \vdots & \iddots & \vdots \\ 1 & W_N^{-(N-1)} & \cdots & W_N^{-(N-1)^2} \end{bmatrix} X \qquad (6)$$

B. *Discrete Wavelet Transform*

The inner product is sum of element-by-element multiplication of two vectors. This means result of inner product is a scalar quantity. A sequence represents a signal or vector. The inner product of two given sequences, $g_n = \{g_0, g_1, \dots, g_{N-1}\}$ and $h_n = \{h_0, h_1, \dots, h_{N-1}\}$ is given by

$$< g_n, h_n >= \sum_n g_n h_n = g_0 h_0 + \dots + g_{n-1} h_{n-1} \qquad (7)$$

The convolution of two sequences gn and hn is given by

$$h * g = \sum_{k \in Z} h_k g_{n-k} = \sum_{k \in Z} g_k h_{n-k} \qquad (8)$$

The *J*-level orthogonal DWT of a sequence xn and IDWT are given by

$$\alpha_k^{(J)} = \sum_{n \in Z} x_n g_{n-2^J k}^{(J)}, k \in Z \qquad (9)$$

$$\beta_k^{(l)} = \sum_{n \in Z} x_n h_{n-2^l k}^{(l)}, l \in \{1,2,\dots,J\} \qquad (10)$$

$$x_n = \sum_{k \in Z} \alpha_k^{(J)} g_{n-2^J k}^{(J)} + \sum_{l=1}^{J} \sum_{k \in Z} \beta_k^{(l)} g_{n-2^l k}^{(l)} \qquad (11)$$

where $g^{(J)}$ and $h^{(l)}$ are called scaling sequence and wavelets, respectively. The $\alpha^{(J)}$ is called coarse projection or scaling coefficients whereas $\beta^{(l)}$ are called finer detail projections or wavelet coefficients. For 3-level, i.e., $J = 3$, both scaling and wavelet coefficients using (9) and (10) are given by

$$\beta_k^{(1)} = \sum_{n \in Z} x_n h_{n-2^1 k}^{(1)} =< x_n, h_{n-2^1 k}^{(1)} >_n$$

$$\beta_k^{(2)} = \sum_{n \in Z} x_n h_{n-2^2 k}^{(2)} =< x_n, h_{n-2^2 k}^{(2)} >_n$$

$$\beta_k^{(3)} = \sum_{n \in Z} x_n h_{n-2^3 k}^{(3)} =< x_n, h_{n-2^3 k}^{(3)} >_n$$

$$\alpha_k^{(3)} = \sum_{n \in Z} x_n g_{n-2^3 k}^{(3)} =< x_n, g_{n-2^3 k}^{(3)} >_n$$

The complete set of basis sequences for 3-level is given by

$$\Phi = \{h_{n-2^1 k}^{(1)}, h_{n-2^2 k}^{(2)}, h_{n-2^3 k}^{(3)}, g_{n-2^3 k}^{(3)}\}_{k \in Z}$$

The Haar basic sequences at J-level are given by

$$g_n^{(J)} = \frac{1}{2^{J/2}} \sum_{k=0}^{2^J-1} \delta_{n-k} \qquad (12)$$

$$h_n^{(l)} = \frac{1}{2^{l/2}} \left(\sum_{k=0}^{2^{l-1}-1} \delta_{n-k} - \sum_{k=2^{l-1}}^{2^l-1} \delta_{n-k} \right) \qquad (13)$$

where δ_n is Kronecker delta sequence and is given by

$$\delta_n = \begin{cases} 1, & n = 0 \\ 0, & otherwise \end{cases}$$

For 3-level, Haar basis sequences using (12) and (13) are given by

$$h_n^{(1)} = \frac{1}{\sqrt{2}}\left(\delta_n - \delta_{n-1}\right)$$

$$h_n^{(2)} = \frac{1}{2}\left(\delta_n + \delta_{n-1} - \delta_{n-2} - \delta_{n-3}\right)$$

$$h_n^{(3)} = \frac{1}{2\sqrt{2}}\left(\delta_n + \ldots + \delta_{n-3} - \delta_{n-4} - \cdots - \delta_{n-7}\right)$$

$$g_n^{(3)} = \frac{1}{2\sqrt{2}}\left(\delta_n + \ldots + \delta_{n-7}\right)$$

In following sections, above equations are implemented in the Python computer programming language.

III. PRELIMINARY IMPLEMENTATION

The Numpy module provides N-dimensional array data type, called ND array. The Python program code developed to implement the circular convolution is shown in Figure 1. It starts with comment lines which begin with number sign (#). The two code lines starting with import keyword find and initialize Python modules Numpy and Matplotlib.pylab and assign local names np and plt respectively. The Matplotlib [9] is a library for creating 2D plots in Python with pylab interface i.e., Matplotlib.pylab which provides functions similar to MATLAB.

The function *comp_circ_conv* computes circular convolution and plots input sequence x_n, filter h_n, and resulting circular convolution $(Hx)_n$ by calling the function *draw_seqn_and_filt*. In function *comp_circ_conv,* Python built-in function *len* is used to find length L and M of filter h_n and input sequence x_n respectively. This step is required to determine the value N which is used for zero-padding both input tsequences.

The variables filtN, seqnN, and Hx are defined as N-point numpy.ndarray sequences. These sequences are initialized with all elements set equal to zero. Both filtN and seqnN are assigned filt and seqn sequences up to length of filt, L, and length of seqn, M, respectively. This is accomplished using statement *filtN[:L] = filt[:]* which assigns first L members of filtN array to members of filt array. In this way, elements of filtN array are members of filt array from 0 to L-1, and remaining members are zeros from L to N-1. This is repeated for seqnN which after assignment has members of seqn from 0 to M-1, and zeros from M to N-1. Note, the indexing in Python follows C/C++ which starts from 0, instead of 1 as in MATLAB. This means members of an N-point array are accessed using index from 0 to N-1.

In first example, the function *comp_circ_conv* input arrays are x_n = {4, 5, 6, 2} and h_n = {0.5, 2, 0.5} which are assigned to arrays seqn and filt, respectively. These arrays are assigned to new arrays seqnN = {4, 5, 6, 2, 0, 0} and filtN = {0.5, 2, 0.5, 0, 0, 0}, refer Figure 1 for plots of zero-padded input sequences. The filtN array is flipped and rolled by using statements *filtNflip = filtN[::-1,...]* and *filtNflip = np.roll(filtNflip,1)* respectively. The output array filtNflip after execution of these two statements is equal to {0.5, 0, 0, 0.5, 2}. At this point, first row of (2), i.e., {h_0, h_5, … , h_1} for N = 6 is available as filtNflip.

A for-loop implements circular convolution in (1) and (2). The loop index is variable k which takes values in range(N) i.e., from 0 to N-1. Note, the colon operator (:) in the for-loop statement tells Python a code block follows. Indentation is very important and used exclusively in this computer language for identification and execution of code blocks. The concept is similar to use of braces f g in C/C++. After completion of for- loop, Hx holds result of circular convolution.

At this point, the function *draw_seqn_and_filt* is called with three parameters: input sequence x_n, filter hn and circular convolution Hx. The definition of this function is shown in Python code in Figure 1. This code is similar to MATLAB figure generating code. It generates three subplots: (a) input sequence x_n, (b) filter h_n and (c) circular convolution Hx. These subplots are shown in Figure 1.

To use the developed program code in Figure 1, add following three code lines at the end of program. Once program code executes it generates subplots and assigns result of circular convolution to H1 due return *Hx* statement in the function comp_circ_conv.

```
x = np.array([4,5,6,2])
h = np.array([0.5,2,0.5])
H1 = comp_circ_conv(h,x)
```

To compute linear convolution which is equivalent to circular convolution for N ≥ L +M − 1, Numpy provides a function numpy.convolve. This function can be used as follows to compute convolution and verify results of earlier developed program code. It has verified for example presented in this section that both results are same.

Now, a thoughtful look at the code in Figure 1 reveals that most instructions are assignment statements for variables to hold and organize data. The steps to compute the circular convolution are followed which include flip and shift one sequence and at each shift compute sum of product with other sequence. In developing program code, there is a need to zero-pad sequences to compute full convolution. A for-loop performs sequence shift and compute sum of product to generate Hx array.

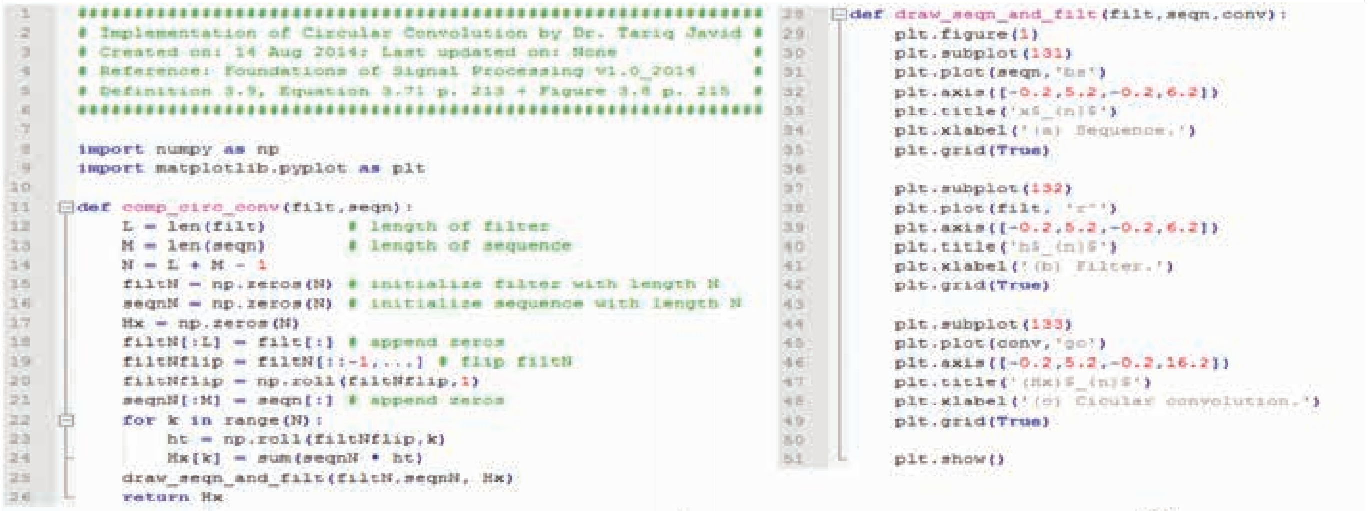
To write a program code for inner product in (7) and make related plots, there is a need to import Numpy and Matplotlib.pyplot. This leads to write two Python import statements below.

```
import numpy as np
import matplotlib.pyplot as plt
```

Above import statements provide functionality available in Numpy module and Pylab Matplotlib interface. The Numpy is short for Numeric Python which provides N-dimensional array functionality to Python basic installation. The Pylab interface is a set of functions in Matplotlib library which provides functionality similar to MATLAB to make 2D plots. An import statement, for example 'import numpy as np' is executed in two steps which are (1) initialize a module, for example numpy, and (2) define a name, for example np.

The inner product of two sequences is computed by defining inprod function as follows. The def keyword is used for function definition. Use of colon at end of a Python statement and indentation at start of a statement are very important for programming in Python. Indentation is used for a block of statements and a colon identifies start of a code block.

```
def inprod(g,h):
    N = len(g)
    tmp = 0
    for n in range(N):
    tmp += g[n] * h[n]
    return tmp
```



```
##################################################################
# Implementation of Circular Convolution by Dr. Tariq Javid #
# Created on: 14 Aug 2014: Last updated on: None             #
# Reference: Foundations of Signal Processing v1.0_2014      #
# Definition 3.9, Equation 3.71 p. 213 + Figure 3.8 p. 215  #
##################################################################

import numpy as np
import matplotlib.pyplot as plt

def comp_circ_conv(filt,seqn):
    L = len(filt)          # length of filter
    M = len(seqn)          # length of sequence
    N = L + M - 1
    filtN = np.zeros(N) # initialize filter with length N
    seqnN = np.zeros(N) # initialize sequence with length N
    Hx = np.zeros(N)
    filtN[:L] = filt[:] # append zeros
    filtNflip = filtN[::-1,...] # flip filtN
    filtNflip = np.roll(filtNflip,1)
    seqnN[:M] = seqn[:] # append zeros
    for k in range(N):
        ht = np.roll(filtNflip,k)
        Hx[k] = sum(seqnN * ht)
    draw_seqn_and_filt(filtN,seqnN, Hx)
    return Hx

def draw_seqn_and_filt(filt,seqn,conv):
    plt.figure(1)
    plt.subplot(131)
    plt.plot(seqn,'bs')
    plt.axis([-0.2,5.2,-0.2,6.2])
    plt.title('x$_{n}$')
    plt.xlabel('(a) Sequence.')
    plt.grid(True)

    plt.subplot(132)
    plt.plot(filt, 'r^')
    plt.axis([-0.2,5.2,-0.2,6.2])
    plt.title('h$_{n}$')
    plt.xlabel('(b) Filter.')
    plt.grid(True)

    plt.subplot(133)
    plt.plot(conv,'go')
    plt.axis([-0.2,5.2,-0.2,16.2])
    plt.title('(Hx)$_{n}$')
    plt.xlabel('(c) Cicular convolution.')
    plt.grid(True)

    plt.show()
```

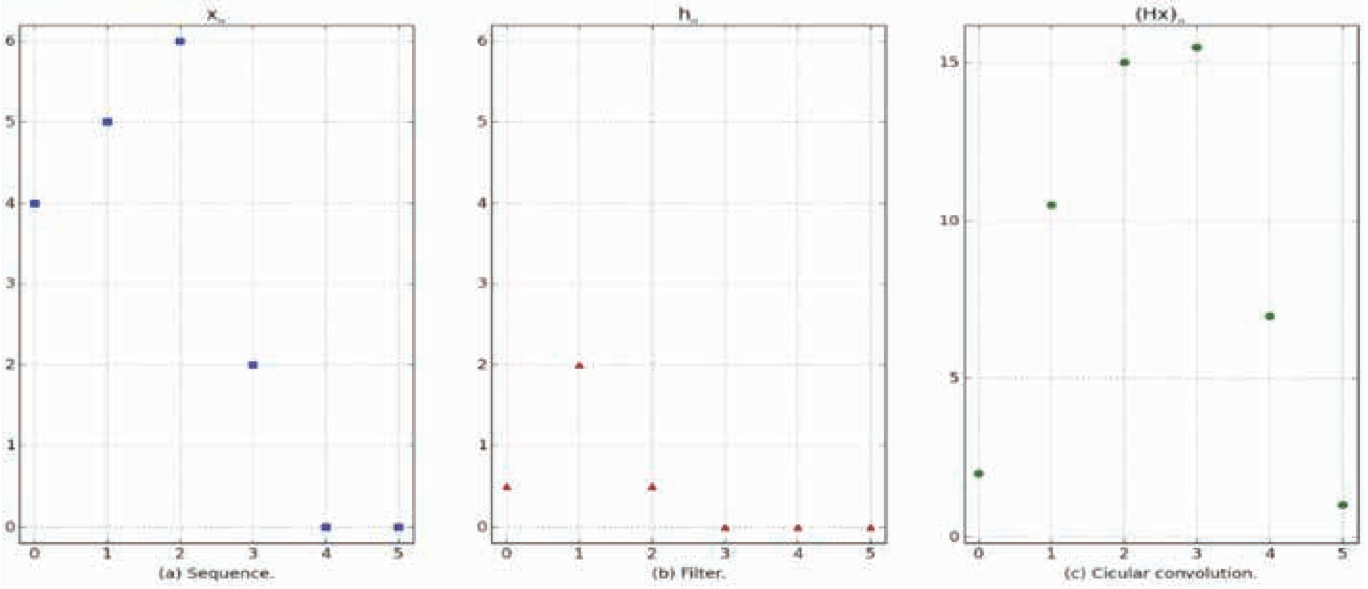


Fig. (1). The code implements circular convolution in Python. The function comp circ_conv computes circular convolution and the function draw seqn and filt plots: (a) input sequence xn = {4, 5, 6, 2}, (b) filter hn = {0.5, 2, 0.5}, and (c) output convolution (Hx)n = {2, 10.5, 15, 15.5, 7, 1}.

In above code, the inprod function takes two sequences g and h as input parameters. A variable N is assigned the length of sequence g. Another variable tmp is initialized with a value of zero. A for loop is used to compute inner product which assigns sum of element-by-element product to tmp variable. After completion of this for loop, tmp contains the result. This function exits by return tmp statement.

To use this function consider an example in which two input sequences are gn = {0, 1, 2, 3, 4, 5} and hn = {5, 4, 3, 2, 1, 0}. These sequences are initialized as two arrays s1 and s2, and function inprod is called with s1 and s2 as parameters. The function inprod returns inner product value which is assigned to a variable inp. This is accomplished by following program code. This program code when executed in Python interpreter displays 20. The print keyword is used to displays the result. Note, the comments in Python begin with a number (#) sign.

```
s1 = np.array([0,1,2,3,4,5])
s2 = np.array([5,4,3,2,1,0])
inp = inprod(s1,s2)
print inp # result is 20
```

The program code to make plots of both input signals and element-by-element multiplication is as follows. The result of execution is shown in Figure 2 which has subplots similar to MATLAB figure with obvious coding similarity.

```
plt.figure(1)
plt.subplot(131)
plt.plot(s1,'ro--')
plt.axis([-0.5, 5.5,-0.5,6.5])
plt.xlabel('Sequence, $s1_{n}$.')
plt.grid(True)
plt.subplot(132)
plt.plot(s2,'bo--')
plt.axis([-0.5, 5.5,-0.5,6.5])
plt.xlabel('Sequence, $s2_{n}$.')
plt.grid(True)
plt.subplot(133)
plt.plot(s1*s2,'go--')
plt.axis([-0.5, 5.5,-0.5,6.5])
plt.xlabel('Sequence, $s1_{n}*s2_{n}$.')
plt.grid(True)
plt.show()
```

The heart of DWT computation lies in the convolution operation between two input signals g_n and h_n. This leads to write a program code for (8) which is accomplished by the function *comp_conv* defined in Python as follows. The comments show four steps of convolution operation which are flip, shift, multiplication, and addition.

```
def comp_conv(g,h):
        L = len(g)
        M = len(h)
        N = L + M - 1
        gn = np.zeros(N)
        gn[:L] = g[:]
        hn = np.zeros(N)
        hn[:M] = h[:]
        hn = hn[::-1,...] # flip
        hn = np.roll(hn,1) # shift
        cnv = np.zeros(N)
        # sum-of-product
        for k in range(N):
            hk = np.roll(hn,k)
            cnv[k] = sum(gn * hk)
        return cnv
```

The above program code is used to compute convolution for earlier signals s1 and s2. The resulting sequence is {0, 5, 14, 26, 40, 55, 40, 26, 14, 5, 0} which is same as computed by the *numpy.convolve* operator. The associated code for this comparison is as follows.

```
np.convolve(s1, s2)
```

Haar basis sequences in (12) and (13) are example basis sequences. These sequences have orthogonal property which means these signals are at right angle to each other. In other words, $<g_n, g_{n-2k}> = <h_n, h_{n-2k}> = \delta_k$ and $<g_n, h_{n-2k}> = 0$. The Python program code to generate level-J transformation matrix is given below.

```
J = raw_input('Input J: ')
J = int(J) # Convert to integer
l = np.array(np.arange(J) + 1)
# Sequence: g_{n}^{J}
gnJ = np.array(np.zeros(2 ** J))
# Sequence: h_{n}^{l}, l = {1,...,J}
hnl = np.array(np.zeros(J*(2**J)))
hnl = hnl.reshape(J,(2**J))
# Compute transformation matrix, T
for k in np.arange(2 ** J):
    gnJ[k] = 2 ** (-J / 2.0)
        for m in l:
            for k in np.arange(2 ** m):
                if k < (2**(m-1)):
                    hnl[m - 1, k]= (2 ** (-m / 2.0))
            elif k < (2**m) :
                    hnl[m - 1, k]= -(2 ** (-m / 2.0))
T = np.concatenate((hnl, [gnJ]),axis=0)
```

The above code starts by an input prompt for J-level, convert input string value to integer value, and initializes Haar basis sequences $g_n^{(J)}$ and $h_n^{(l)}$. A for loop computes the sequence as per (12). A nested for loop computes the sequence $h_n^{(l)}$ as per (13). Both sequences are concatenated as the transformation matrix T.

The output transformation matrix for scale entered as input J = 3 is given below. Note that the rows of this matrix starting from first row correspond to $h_n^{(1)}, h_n^{(2)}, h_n^{(3)},$ and $g_n^{(3)}$ respectively. The elements of this matrix are shown as fractions instead of real numbes.

$$T = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{-1}{2} & \frac{-1}{2} & 0 & 0 & 0 & 0 \\ \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & \frac{-1}{2\sqrt{2}} & \frac{-1}{2\sqrt{2}} & \frac{-1}{2\sqrt{2}} & \frac{-1}{2\sqrt{2}} \\ \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} \end{bmatrix}$$

IV. IMPLEMENTATION OF DFT

In this section, the DFT and IDFT as defined in (4) and (6) are implemented in Python and applied on following example input sequences to compute a length-16 DFT.

$$x_n, y_n = \cos\left(\frac{2\pi}{16}n\right), \cos\left(\frac{2\pi}{32}n\right) \qquad (14)$$

In the start of program code development for DFT, there is a need to import Numpy and Matplotlib.pyplot. This leads to write two Python import statements below. These instructions provide functionality available in Numpy Python module and Pylab Matplotlib interface.

```
import numpy as np
import matplotlib.pyplot as plt
```

Once import of required modules is accomplished, the next step is to define and initialize variables appropriate data types. This is done by following three statements. In this code, to compute length-16 DFT, a constant value 16 is assigned to N. As Python is a case sensitive language similar to C/C++, therefore N and n are two different data variable names. The data type array is used and Numpy buil-in function numpy.arange assigns n = {0, 1, ... , 15}. Note, np.arange is used instead of numpy.arange due local name np is assigned to numpy in earlier import statement. Last statement in this code initializes x and y with sequences in (14). This statement combines two statements in one instruction. Python integer division is different than float division, to use float division 16.0 is typed instead of 16 and similarly 32.0 is used instead of 32.

```
N = 16 # Data type: integer
# Data type: numpy.ndarray
# n = [0,1,...,15]
n = np.arange(N)
x, y = np.cos(2 *np.pi*n/16.0), np.sin(2*np.pi*n/32.0)
```

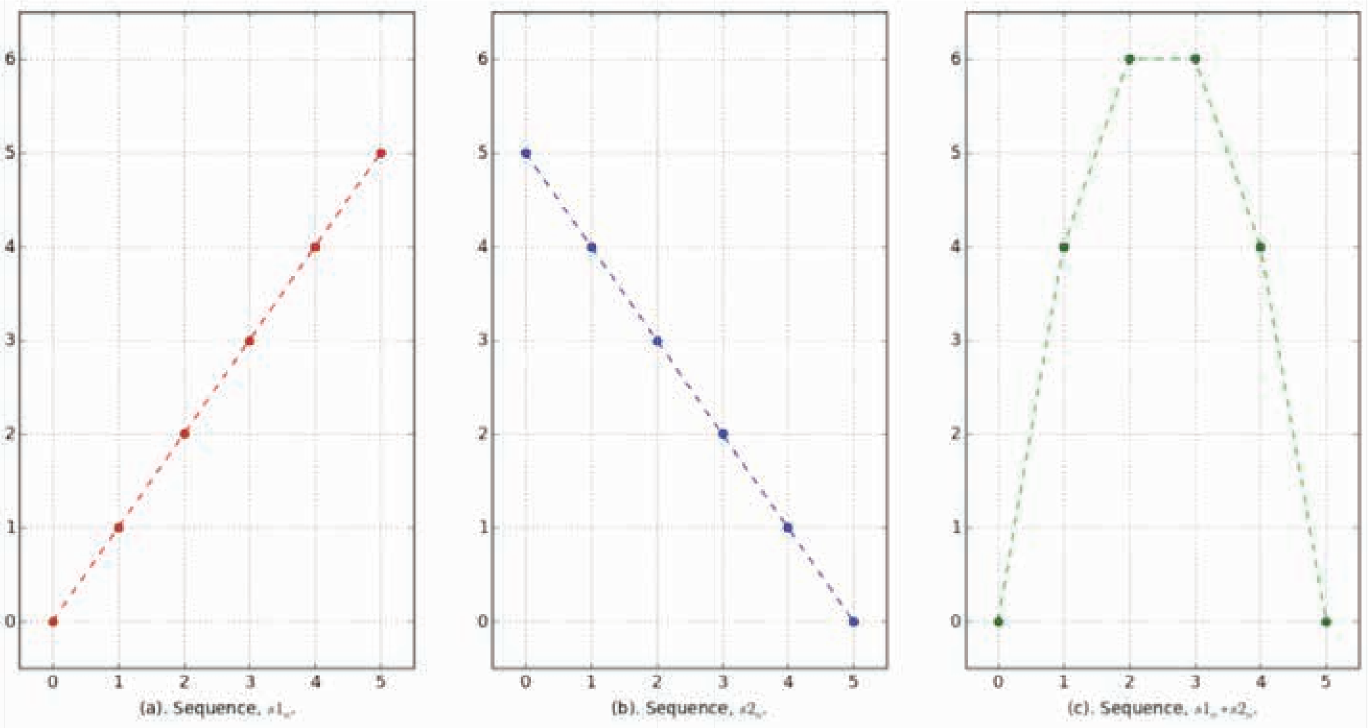


Fig. (2). Plots of sequences (a) $s1_n = \{0, 1, 2, 3, 4, 5\}$, (b) $s2_n = \{5, 4, 3, 2, 1, 0\}$, and (c) s1 * s2 which leads to < s1, s2 > = ∑(0, 4, 6, 6, 4, 0) = 20.

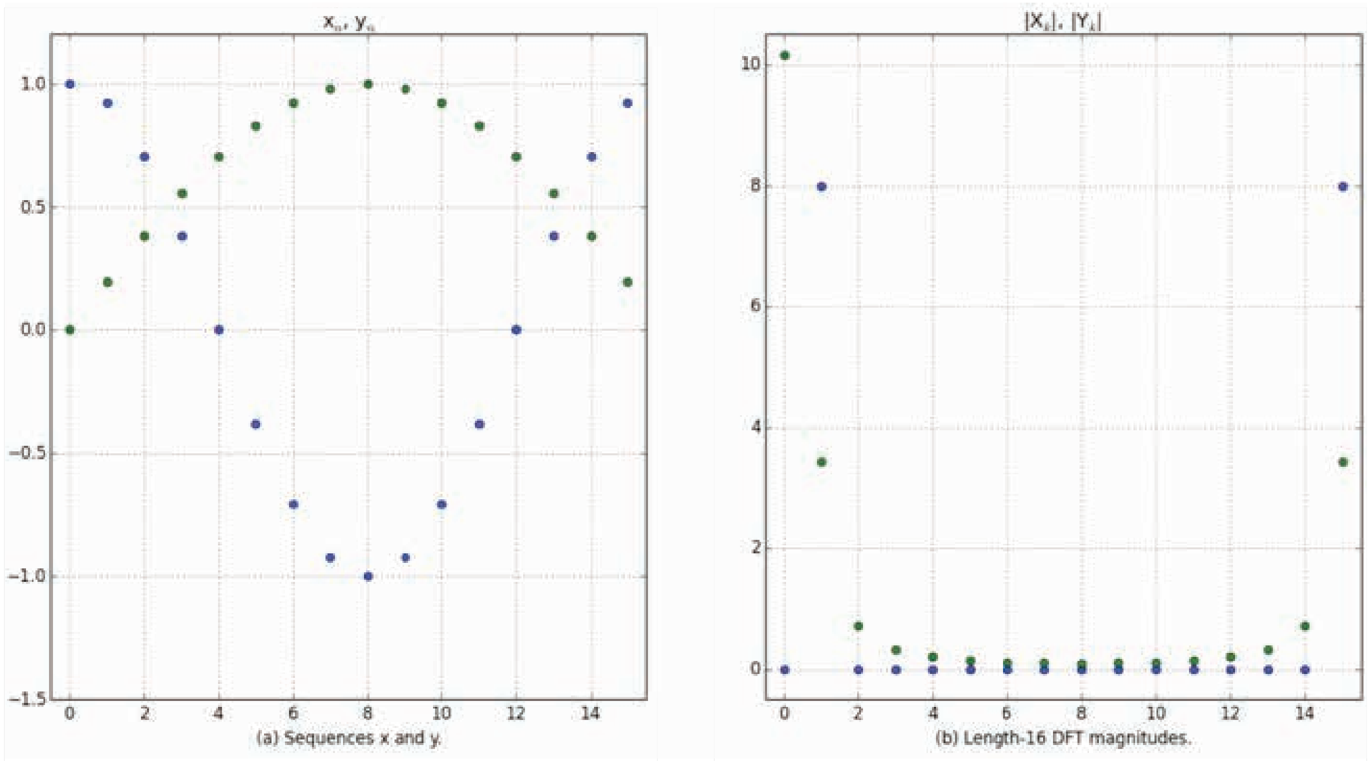


Fig. (3). Plots of (a) sequences x_n and y_n in (14) and (b) length-16 DFT magnitudes $|X_k|$ and $|Y_k|$.

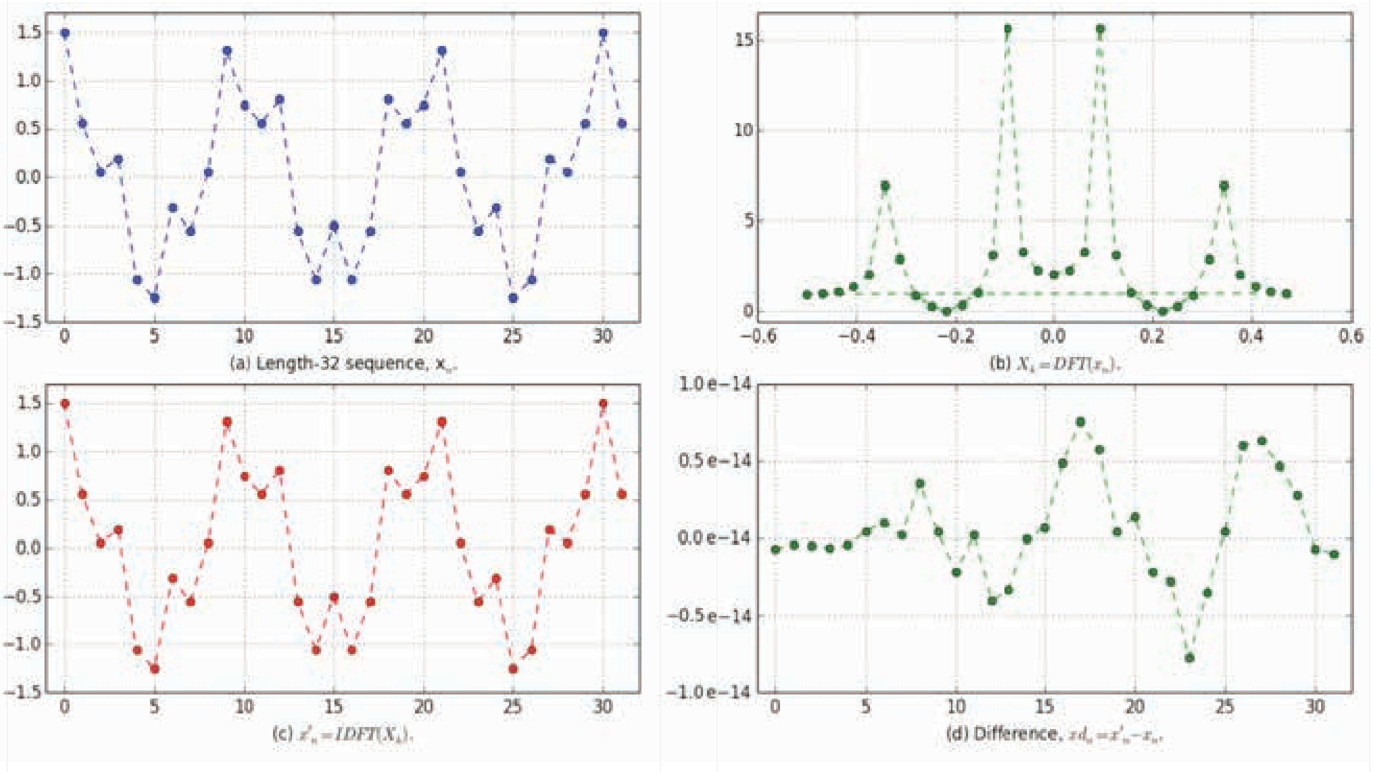


Fig. (4). Plots of (a) sequence x_n in (15), (b) length-32 DFT magnitude $|X(ej^w)|$ using Numpy built-in function numpy.fft.fft to compute DFT, (c) sequence, x'_n = IDFT(Xk) using developed program code in Section IV, and (d) the difference sequence, $xd_n = x'_n - x_n$.

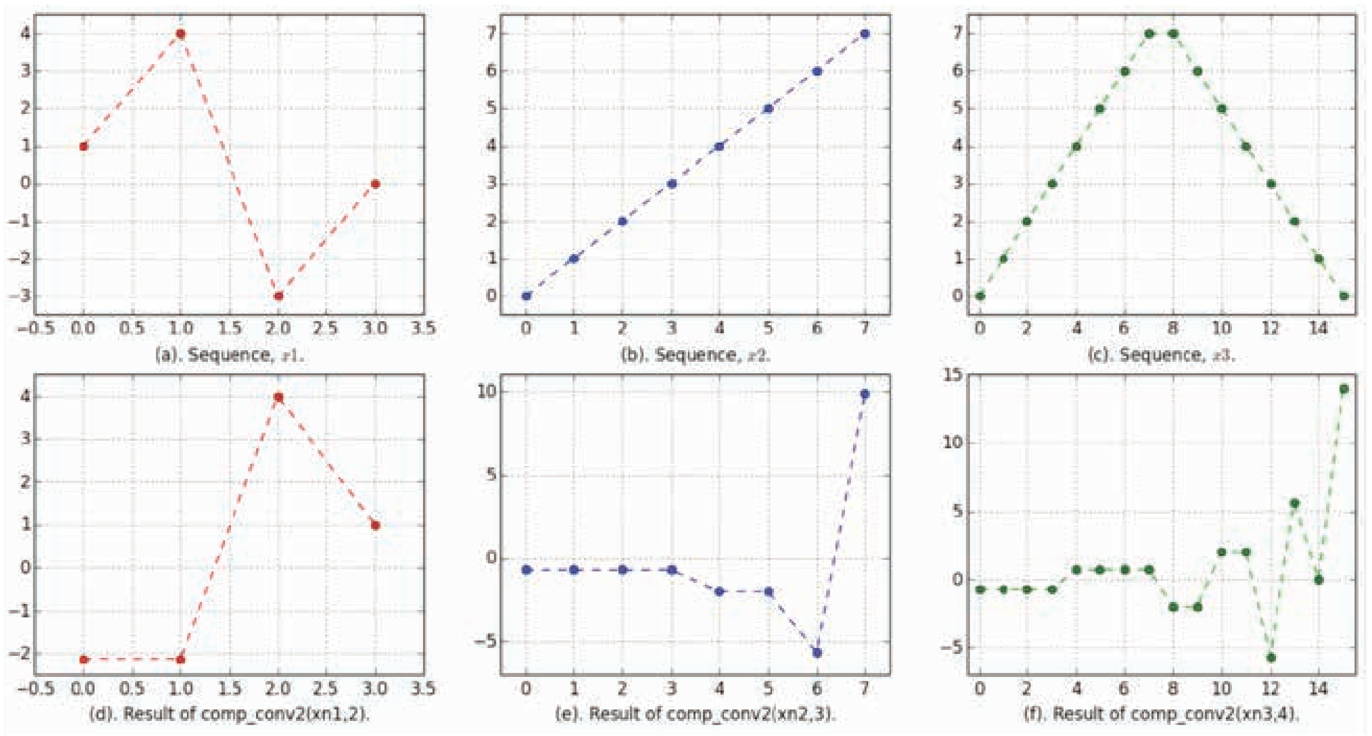


Fig. (5). Plots of sequences and orthogonal DWT: (a) x1 = {1, -4, 3, 0}, (b) x2 = {0, 1, 2, 3, 4, 5, 6, 7}, (c) x3 = {0, 1, 2, 3, 4, 5, 6, 7, 7, 6, 5, 4, 3, 2, 1, 0} and (d)-(f) results of *comp_conv2(sequence,level)*. The orthogonal DWT coefficients are organized as $\{\beta_k^{(1)}, \quad ,\beta_k^{(J)},\alpha_k^{(J)}\}$.

The next part of DFT program code initializes arrays for intermediate operations and to hold final results. The F matrix is an NxN array which corresponds to (4), X and Y are for DFT of x and y, and Xmod and Ymod are for magnitudes |X| and |Y|, respectively.

```
F = np.zeros((N,N),dtype=np.complex)
X = np.zeros(N,dtype=np.complex)
Y = np.zeros(N,dtype=np.complex)
Xmod = np.zeros(N,dtype=np.float)
Ymod = np.zeros(N,dtype=np.float)
```

The rest of program code to implement DFT is given below. In this code, the value of unit-modulus Eigen sequence, i.e., $W_N^{kn} = e^{-j(2\pi/N)kn}$ is computed first which is required to compute (4). A nested for loop is used in the program code. The inner for loop is used to compute elements of each row of F matrix and outer for loop is used to compute DFT coefficients. After execution of both for loops, X and Y arrays hold DFT results. Each element of both arrays is a complex number. The relation $|X| = \sqrt{(\text{Re})^2 + (\text{Im})^2}$ is used to compute Fourier spectrum of X i.e. magnitude of X or |X|. The same is repeated to compute |Y|. Figure 3 shows subplots related to this example input sequence (14).

```
WN = np.exp(-2.0j * np.pi / float(N))
for k in range(N):
    for m in range(N): # m = n
        pwr = k * m
        F[k,m] = WN ** pwr
    X[k] = sum(F[k,:] * x[:])
    Y[k] = sum(F[k,:] * y[:])
Xmod = np.sqrt(X.real ** 2 + X.imag ** 2)
Ymod = np.sqrt(Y.real ** 2 + Y.imag ** 2)
```

In above part of program code for DFT in Python, use of *float(N)* in computing value of WN shows a type conversion from integer type to float type. This is required so that Python interpreter uses float division instead of integer division. To understand this, try execution of following code in Python interpreter. The integer division i.e. 1/2 results in 0 whereas the float division, i.e., *1/float(2)* results in 0.5. This is an important point while coding fractions or working with real numbers in Python.

```
>>> 1/2
0
>>> 1/float(2)
0.5
```

Array slicing is used to multiply a row of F and x inside the function sum. This means multiplication expression F[k,:] * x[:] effectively generates element-by-element product of kth row of F matrix and input sequence x. Note that power or exponent operator in Python is ** which means expression

X.real ** 2 computes $(\text{Re})^2$. Finally, np.sqrt calculates square root. Now, to verify the program code developed in this section generates similar results as Numpy built-in FFT function, use following statement.

```
X = np.fft.fft(x,N)
```

The program code developed for DFT in Section IV can be modified to compute IDFT. These changes are: compute W_N^{-1}, instead of WN, and divide it by N, recall float division.

This code is given below.

```
WN = np.exp(2.0j * np.pi / float(N))
    ...
    x[n] = sum(F[n,:] * X[:]) / float(N)
    y[n] = sum(F[n,:] * Y[:]) / float(N)
    ...
```

In above code, index variable n is used instead of k. The built-in function in Numpy for IDFT is numpy.fft.ifft.
To demonstrate application of program code developed for IDFT, consider input sequence xn below for n = {0, … , 31}. Length-32 DFT i.e. X_k is computed for k = {0, … , 31} using numpy.fft.fft function, See Figure 4.

$$x_n = \cos\left(\frac{2\pi}{10}n\right) + \frac{1}{2}\cos\left(\frac{2\pi}{3}n\right) \tag{15}$$

The length-32 IDFT is computed using program code developed in this section and result is plotted in Figure 4 along with input example sequence in (15). There are four subplots in this figure. The subplot (a) shows input sequence for length-32, (b) shows length-32 DFT of input sequence using Numpy built-in function numpy.fft.fft, the program code developed in Section IV for DFT can also be used to generate same output, (c) shows length-32 IDFT to generate time-domain sequence, x'_n, and (d) shows difference sequence, xd_n of sequences x_n and x'_n. A very small or negligible difference which is in the order of 10^{-14} indicates that developed program code for IDFT in this section produces similar results.

V. IMPLEMENTATION OF DWT

In this section, orthogonal DWT and IDWT, as defined in (9), (10), and (11) are implemented in Python. The developed program code uses the Haar basis sequences in (12) and (13); and is implemented as the transformation matrix explained earlier. The program code is given below as function definition comp_conv2. It is a variant of earlier developed comp_con function.

```
def comp_conv2(xin,lvl):
    T = comp_trans(lvl)
    n = T.shape[0]
        d = T.shape[1]
        conv2 = np.array(np.zeros(2 ** lvl))
        tmp, idx = 0, 0
        # coefficients: 1 to level-(J-1)
            for k in range(n-2):
            for m in range((d / (2**(k+1)))):
            row = np.roll(T[k,:],(2*m)*(2**k))
            tmp = sum(row * xin)
            conv2[idx] = tmp
            idx = idx + 1
# compute level-J wavelet and approximation coefficients
    betaJ = sum(T[lvl-1,:] * xin)
    alphaJ = sum(T[lvl,:] * xin)
    # complete transformation
    conv2[d-2],conv2[d-1] = betaJ,alphaJ
    return conv2
```

The above program code is used to compute orthogonal DWT of following sequences x1, x2, and x3 for levels *J = 2, J = 3,* and J = 4, respectively.

```
x1 = {1, 4, -3, 0}
x2 = {0, 1, 2, 3, 4, 5, 6, 7}
x3 = {0, 1, 2, 3, 4, 5, 6, 7, 7, 6, 5, 4, 3, 2, 1, 0}
```

The resulting DWT coefficients, organized as $\{\beta_k^{(1)},\ldots,\beta_k^{(J)},\alpha_k^{(J)}\}$, are given below. These results are same as computed with Python module PyWavelets. Figure 5 shows plots of sequences and orthogonal DWT. The elements of resulting array are shown up to three decimal places.

```
comp_conv2(xn3,4)
# [-0.707 -0.707 -0.707 -0.707
#   0.707  0.707  0.707  0.707
#   -2.     -2.      2.      2.
#   -5.656  5.656  0.      14. ]
comp_conv2(xn2,3)
# [-0.707 -0.707 -0.707 -0.707
#  -2.      -2.     -5.656   9.899]
comp_conv2(xn1,2)
# [-2.121 -2.121  4.       1.]
```

The developed program code for orthogonal IDWT implementation is given below. The function comp conv3 performs a reverse operation as of earlier function comp conv2. Note, a backslash ‘\’ is used for a multi-line Python statement.

```
def comp_conv3(win,lvl):
    W = win
    T = comp_trans(lvl)
    n = T.shape[0]
    d = T.shape[1]
    conv3 = np.array(np.zeros(2**lvl))
```

```
        temp1 = np.array(np.zeros(2**lvl))
        temp2 = np.array(np.zeros(2**lvl))

    T[lvl,:] = W[d-1] * T[lvl,:]
        T[lvl-1,:] = W[d-2] * T[lvl-1,:]
        tmp, idx = 0, 0
        for k in range(n-2):

    temp1[:], temp2[:] = 0, 0
        for m in range((d/(2**(k+1)))):
            temp1 = np.roll(T[k,:], (2*m)*(2**k))
            temp1 = temp1 * W[idx]
            temp2 += temp1
             idx = idx + 1
        T[k,:] = temp2
    conv3 = sum(T[:,])
    return conv3
```

The above code is used to compute orthogonal IDWT of signals x1, x2, and x3 transformed to Wavelet-domain by using Python function *comp_conv2* for levels $J = 2, J = 3$, and $J = 4$, respectively; and are given below. The transformed pair is $x_n \leftrightarrow x_W^{(J)}$, where W denotes the Wavelet-domain representation.

$$x1_W^{(2)} = \{\frac{-3}{\sqrt{2}}, \frac{-3}{\sqrt{2}}, 4, 1\}$$

$$x2_W^{(3)} = \{\frac{-1}{\sqrt{2}}, \frac{-1}{\sqrt{2}}, \frac{-1}{\sqrt{2}}, \frac{-1}{\sqrt{2}}, -2, -2, \frac{-8}{\sqrt{2}}, \frac{14}{\sqrt{2}}\}$$

$$x3_W^{(4)} = \{\frac{-1}{\sqrt{2}}, \ldots, \frac{1}{\sqrt{2}}, \ldots, -2, -2, 2, 2, \frac{-8}{\sqrt{2}}, \frac{8}{\sqrt{2}}, 0, 14\}$$

To use function comp_conv3 to compute IDWT in program code, define Numpy arrays for $x1_W^{(2)}$, $x2_W^{(3)}$, and $x3_W^{(4)}$ as follows for above Wavelet-domain representation. The statement $x_n = comp\_conv3(x_W^{(J)}, J)$ computes the corresponding orthogonal IDWT. The results are shown as comments.

```
    xn1w2 = np.array([-3/trt,-3/trt,4,1])
    xn2w3 = np.array([-1/trt,-1/trt, \
                    -1/trt,-1/trt,-2,-2,-8/trt,14/trt])
    xn3w4 = np.array([-1/trt,-1/trt, \
                     -1/trt,-1/trt,1/trt,1/trt,1/trt, \
                      1/trt,-2,-2,2,2,-8/trt,8/trt,0,14])
    xn1 = comp_conv3(xn1w2,2)        # [ 1. 4. -3. 0. ]
    xn2 = comp_conv3(xn2w3,3)
        # [ 4.44e-16 1. 2. 3. 4. 5. 6. 7. ]
    xn3 = comp_conv3(xn3w4,4)
    # [ 0. 1. 2. 3. 4. 5. 6. 7. 7. 6. 5. 4. 3. 2. 1. 0.]
```

In above code, the comment for comp_conv3(xn2w3,3) shows zero as 4.44 x 10—16 – a value very near to zero. All other results show successful computation of orthogonal IDWT. Use of PyWavelets results in same values, when applied to xn1w2, xn2w3, and xn3w4. Use following code to compute IDWT by PyWavelets for $x2_W^{(3)}$, i.e., xn2w3.

```
    import pywt # import PyWavelets
    xn2w3_cA3 = np.array([xn2w3[7]])
    xn2w3_cD3 = np.array([xn2w3[6]])
    xn2w3_cD2 = xn2w3[4:6]
    xn2w3_cD1 = xn2w3[0:4]
    xn2w3_wt = [xn2w3_cA3, xn2w3_cD3, \
                xn2w3_cD2, xn2w3_cD1]
    xn2 = pywt.waverec(xn2w3_wt,'haar')
    # [ 4.44e-16 1. 2. 3. 4. 5. 6. 7. ]
```

VI. DISCUSSION

This study aimed to develop a deeper understanding of both DFT and orthogonal DWT by the development of computer program code to implement respective mathematical formulations in Python language. We have followed a step-by-step approach to achieve our research objectives. These objectives include an organized presentation of related mathematical expressions from sources [7] and [8], an introduction of Python computer programming language basics which are directly helpful in the development of the program code, and application on selected example signals for results validation. Also, the results are compared with built-in functions to show the correctness of our implementation.

Almost all famous computer programming languages and mathematical software packages have implemented both DFT and DWT as built-in functions. However, most of these implementations use pre-compiled code to achieve faster execution. This act posed serious limitations towards an in-depth learning of many useful mathematical formulations. The learners in STEM disciplines often encountered mathematical formulations, for example complex transformations, which are important for them to master. The work in this paper has focused on helping learners to master such mathematical formulations with the help of developing code in a computer programming language. Python is rapidly evolving language used by many valuable organizations which are especially active in carrying out scientific research.

We encourage learners to use the program code developed in this study, enhance and modify it as per their needs. As pointed out earlier, this is not the case for the most of computer programming languages and mathematical software packages. They either provide pre-compiled code which cannot be modified or a cumbersome process to alter the code up to a certain level. Further, the program code developed in this study is based on a solid mathematical framework from [7-8]. A number of examples from these valuable resources are used to show application of the developed program code.

An interesting finding of this research study is that we need not to code the complete transformation in a computer programming language. This means while developing a program code, we can focus on either forward transformation or reverse transformation. Once, a forward or reverse transformation is implemented in Python; it can be evaluated on signals and compared with built-in functions. For example, if a forward transformation is developed; the built-in reverse transformation can be used. This is a great flexibility as one may want to work on implementation of forward transformation only.

The implementation described in this research study is primarily based on fundamental mathematical formulations of DFT and orthogonal DWT. Advanced and recent fast computation algorithms are not considered for implementation. A time comparison is also not considered due scripting nature of Python computer programming language.

VII. CONCLUSION

A successful implementation to compute both DFT and orthogonal DWT in Python computer programming language has been presented in this study with application to selected example time-domain sequences. A step-by-step approach in the development of program codes has been followed for successful implementation of circular convolution, DFT, IDFT, DWT, and IDWT. A clear explanation of program code has been made to motivate and attract novice readers to learn fundamental and advanced signal processing tools. The research has been aimed to strengthen understanding of transformation computation steps involved in computing and plotting DFT and DWT. The results have shown a very close agreement with those obtained using built-in functions in Numpy and PyWavelets modules. The future research directions include implementation of biorthogonal Wavelet Transform in Python and extend program code developed in this study to compute higher dimension DFT and DWT.

ACKNOWLEDGEMENT

First authors would like to thank the Coursera MOOC education platform (http://www.coursera.org) and Dr. Charles Severance, Clinical Associate Professor, School of Information, University of Michigan, for the online course titled, “Programming for Everybody (Python),” offered in 2014. An appreciation for reviewers as their remarks helped to improve the quality of work presented in this paper.

REFERENCES

[1] S. Cass. (2016). *The 2016 Top Ten Programming Languages* [Online]. Available: http://spectrum.ieee.org/computing/software/the-2016-top-programming-languages.

[2] P. Guo. (2014). *Python is Now the Most Popular Introductory Teaching Language at Top US Universities* [Online]. Available: http://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-u-s-universities/fulltext.

[3] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach*. New Jersey: Addison-Wesley, 2016.

[4] E. Muller, J. A. Bednar, M. Diesmann, M.-O. Gewaltig, M. Hines, and A. P. Davison, “Python in Neuroscience,” *Frontiers in Neuroinformatics,* vol. 9, pp: 1-4, 2015. http://dx.doi.org/10.3389/fninf.2015.00011.

[5] S. Walt, S. C. Colbert, and G. Varoquaux, “The NumPy Array: A Structure for Efficient Numerical Computation,” *Computing in Science & Engineering,* vol. 13, no. 2, pp: 22-30, 2011. http://dx.doi.org/10.1109/MCSE.2011.37.

[6] *PyWavelets–Discrete Wavelet Transform in Python* [Online]. Available: https://pywavelets.readthedocs.io/en/latest.

[7] M. Vetterli, J. Kovacevic, and V. Goyal, *Foundations of Signal Processing*. United Kingdom: Cambridge University Press, 2014.

[8] J. Kovacevic, V. Goyal, and M. Vetterli, *Fourier and Wavelet Signal Processing*. United Kingdom: Cambridge University Press, 2015.

[9] M. A. Wood, *Python and Matplotlib Essentials for Scientists and Engineers*. United Kingdom: Morgan & Claypool Publishers, 2015.