The Hong Kong Polytechnic University
Department of Electronic and Information Engineering

Digital Signal Processing (EIE4413)
Lab 1: Spectrum Analysis and Filtering

Purpose

This lab explores the implementation of various fundamental concepts in signal processing using Python. The lab is divided into two parts: (i) DFT and FFT; and (ii) LTI Systems and Filtering. They serve as the supplement to the discussion in class.

Things to do

For each question in this lab sheet, give your answer right under the question. Submit this lab sheet (with your answers) through the Blackboard to your lecturer before the deadline. In addition, show the result of Q.2.5 to your tutor during the laboratory session.

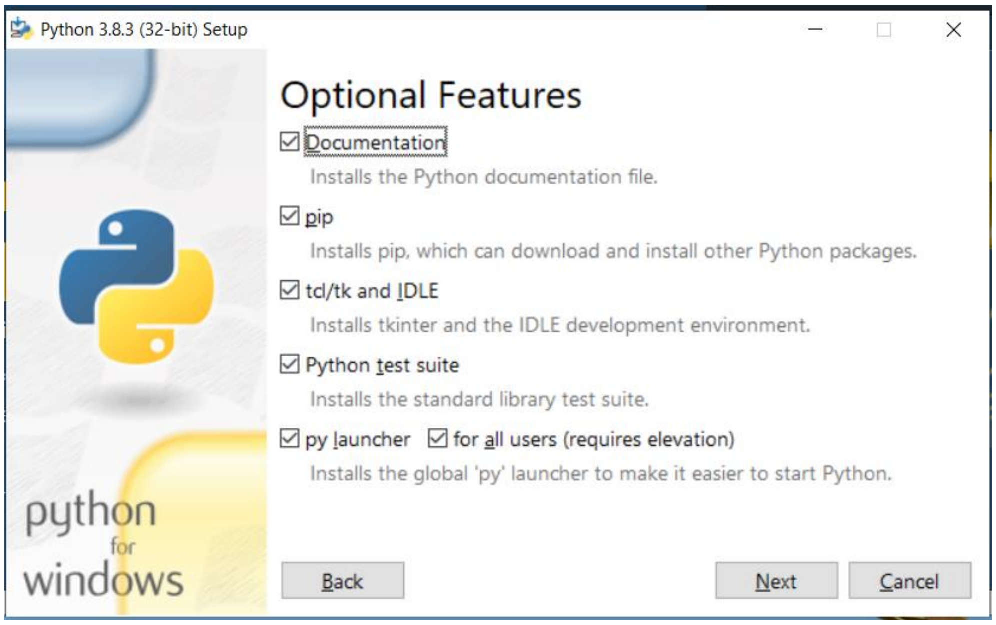
Equipment

PC with Windows 7 or above
PC with PyCharm IDE tools 2020.1 or above
PC with Python 3.8 or above

Part 0. Installing Python and required libraries

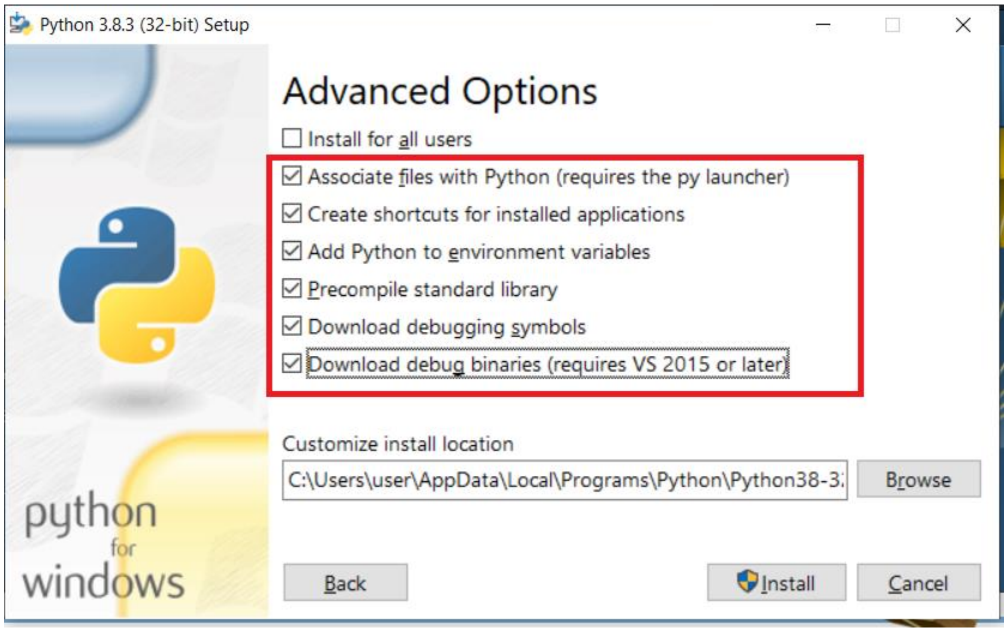
Python is a general-purpose programming language created by Guido van Rossum in 1991. It has been one of the most popular programming languages nowadays. To write Python programs for this lab exercise, you need to install Python, PyCharm IDE on your computer. If you are using the computers in the lab, these tools are already installed in the computers there.

The latest Python can be downloaded at https://www.python.org/downloads/. When running Python installation, select all these items:


Python 3.8.3 (32-bit) Setup

Optional Features

- [✓] Documentation
  Installs the Python documentation file.
- [✓] pip
  Installs pip, which can download and install other Python packages.
- [✓] tcl/tk and IDLE
  Installs tkinter and the IDLE development environment.
- [✓] Python test suite
  Installs the standard library test suite.
- [✓] py launcher [✓] for all users (requires elevation)
  Installs the global 'py' launcher to make it easier to start Python.


Python 3.8.3 (32-bit) Setup

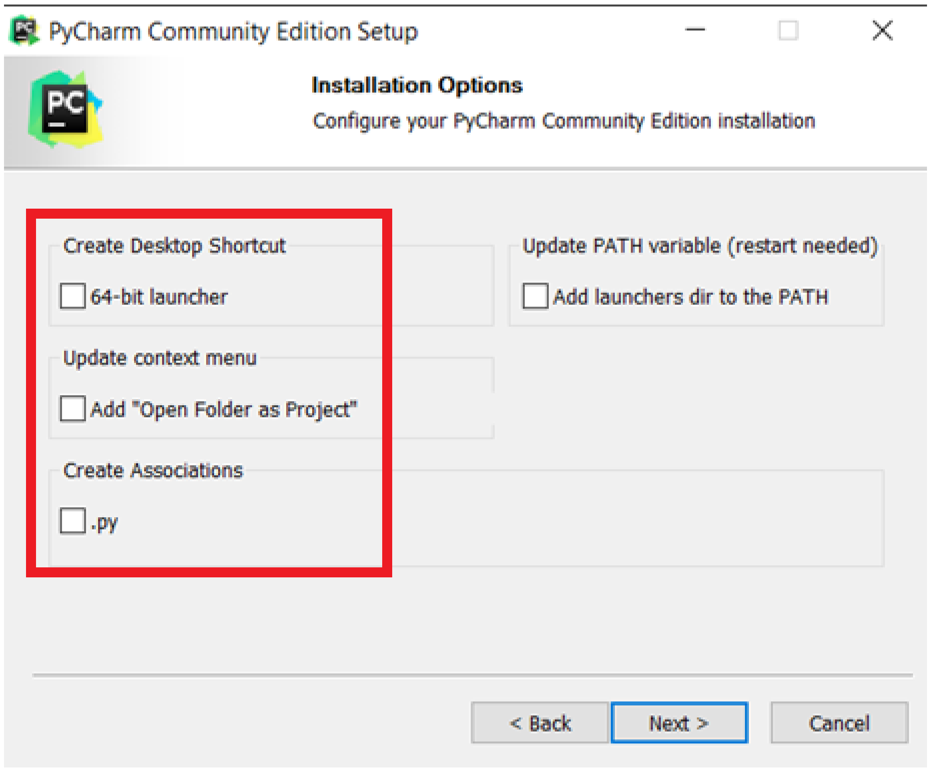
Advanced Options

- [ ] Install for all users
- [✓] Associate files with Python (requires the py launcher)
- [✓] Create shortcuts for installed applications
- [✓] Add Python to environment variables
- [✓] Precompile standard library
- [✓] Download debugging symbols
- [✓] Download debug binaries (requires VS 2015 or later)

Customize install location
C:\Users\user\AppData\Local\Programs\Python\Python38-3:

The PyCharm IDE is an integrated development environment for developing Python applications. It can be run on Windows, macOS and Linux. For downloading PyCharm IDE, click https://www.jetbrains.com/pycharm/download/ and follow the instruction to complete the installation. The community version of PyCharm is free. If you want to have a try of PyCharm, the community version is sufficient.

Note that when installing PyCharm, the following options should be selected.



PyCharm Community Edition Setup

Installation Options

Configure your PyCharm Community Edition installation

Create Desktop Shortcut

☐ 64-bit launcher

Update PATH variable (restart needed)

☐ Add launchers dir to the PATH

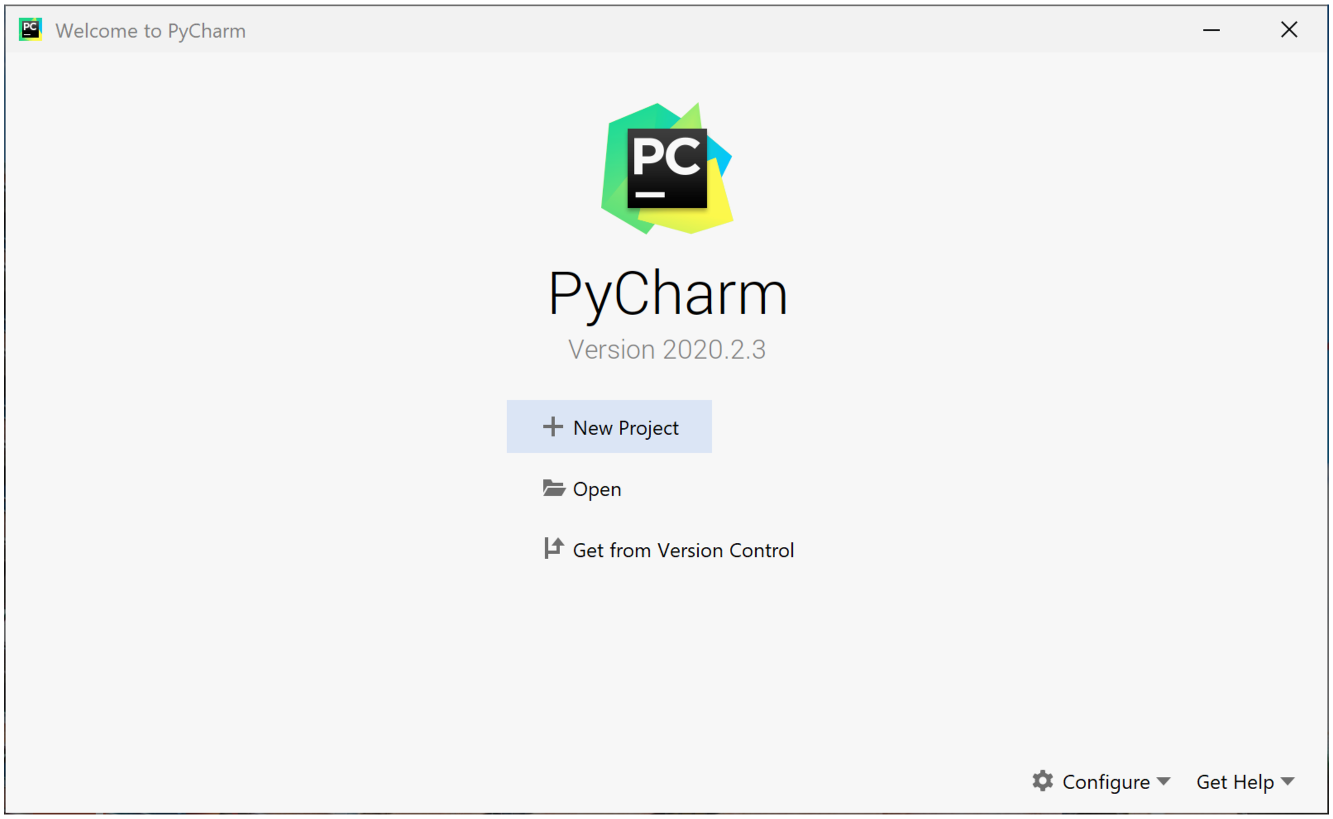
Update context menu

☐ Add "Open Folder as Project"

Create Associations

☐ .py

After installing PyCharm, start PyCharm and click New Project.



Welcome to PyCharm

PyCharm

Version 2020.2.3

If you are doing this lab exercise with the computer in the lab, start PyCharm and click the New Project button. Then on the window that pops up, enter your "*home direction*\Lab1Project" in Location and click Create.

For this lab exercise, you need to use the library numpy and matplotlib. You need to install them. Click Terminal at the bottom of the window. Enter the command:

```
pip install numpy
```

Then, enter the command:

```
pip install matplotlib
```

After installed all the required libraries. You are now ready to write programs for this lab.

Part 1. DFT and FFT

The Discrete Fourier Transform (DFT) allows us to evaluate the Fourier Transform of a signal at *N* evenly spaced frequencies, where *N* is the number of data in that signal. Mathematically, the DFT of a signal $x[n]$, where $n = 0, 1, \ldots, N-1$, is defined as follows:

$$X[k] = \sum_{n=0}^{N-1} x[n]e^{-i2\pi nk/N}$$

where $k = 0, 1, \ldots, N-1$. Basically, the number of complex multiplications and additions required by an *N*-point DFT is N^2 and *N(N-1)*, respectively. It becomes a heavy computational burden when *N* is large. The FFT is then devised to reduce the computational complexity, however it has the limitation that *N* must be a power 2.

In Python, the method of how a DFT is computed is determined by the value of *N*. If *N* is a power of 2, FFT will be used. If *N* is, for instance, a prime number, the traditional DFT will be used. To examine their computational complexity, we can use the `time` function to measure the time elapsed to execute the FFT.

Q.1.1 Try the program above. Show the difference in time taken when calculating the FFT of *x* and y.

Procedure:

1. Add the following function `part1()` to the program main.py created by the system and then call the function in main. Note that the indentation in the program is important.

```
2. def part1():
       import numpy    # import numpy library to use FFT and arrays
       from numpy import random    # import numpy random function
       import time  # time()

       # Q1.1
       x = random.rand(65536)
       y = random.rand(65213)
       tic = time.time()  # count start time
```

```
        x1 = numpy.fft.fft(x)
        toc = time.time()  # count finish time
        print('FFT(x)=', (toc-tic))  # time elapse
        xtime=(toc-tic)

        tic = time.time()  # count start time
        y1 = numpy.fft.fft(y)
        toc = time.time()  # count finish time
        print('FFT(y)=', (toc-tic))  # time elapsed
        ytime=(toc-tic)

        timedifference=numpy.abs(ytime-xtime)
        print('FFT(|x-y|)=', timedifference)
```

3. Run the program by pressing the ▶ button at the top right corner of the PyCharm IDE.
4. Since there are many factors affecting the timing measurement, you should run the program a few times and take the average as the final result.

Question:
(a) Compare the average running times in both cases.

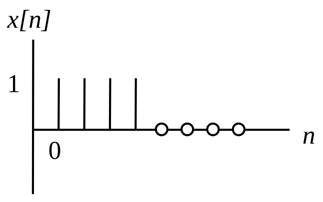
0.007997	0.0080003	0.0080034	0.0080001	
0.025002	0.0229976	0.0240016	0.0230026	

Average t: FFT(x) = 0.0080002; FFT(y) = 0.0237510. Difference = 0.0157508

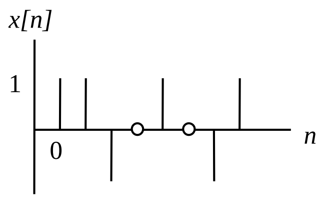
Q.1.2 With reference to the above program, find the 8-point FFTs of each of the four signals as shown below:

Procedure:
1. Add the following function `part1_2()` to the program main.py and then call the function in main.
2. Write the codes for computing the FFT of the following 4 data sequences.

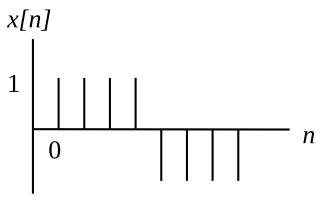
```
def part1_2():
    import numpy as np
    np.set_printoptions(precision=2, linewidth=100)

    # Q1.2, find 8-point FFT of 4 signals
    x1 = np.array([1, 1, 1, 1, 0, 0, 0, 0])
    x2 = np.array([1, 1, -1, 0, 1, 0, -1, 1])
    x3 = np.array([1, 1, 1, 1, -1, -1, -1, -1])
    x4 = np.array([0, 1, 1, 1, 0, -1, -1, -1])

    h1 = np.fft.fft(x1,8)
    h2 = np.fft.fft(x2, 8)
    h3 = np.fft.fft(x3, 8)
    h4 = np.fft.fft(x4, 8)

    print('\n', h1, '\n\n', h2, '\n\n', h3, '\n\n', h4)
```



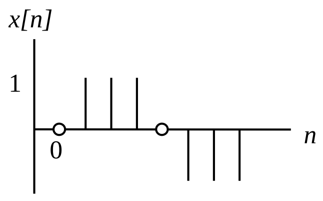
(A)



(B)



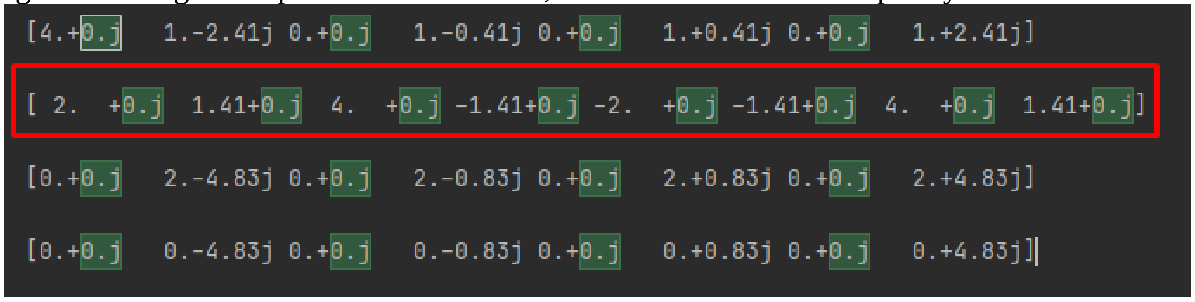
(C)



(D)

Questions:

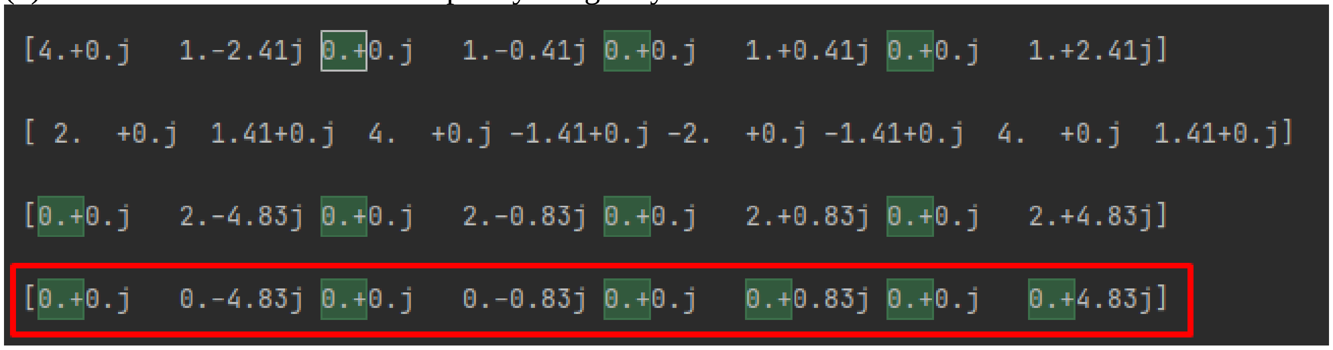
(a) Ignore the slight computer round-off error, which of these FFTs are purely real?



```
[4.+0.j    1.-2.41j 0.+0.j    1.-0.41j 0.+0.j    1.+0.41j 0.+0.j    1.+2.41j]

[ 2.   +0.j   1.41+0.j   4.   +0.j -1.41+0.j -2.   +0.j -1.41+0.j   4.   +0.j   1.41+0.j]

[0.+0.j    2.-4.83j 0.+0.j    2.-0.83j 0.+0.j    2.+0.83j 0.+0.j    2.+4.83j]

[0.+0.j    0.-4.83j 0.+0.j    0.-0.83j 0.+0.j    0.+0.83j 0.+0.j    0.+4.83j]
```

B.

(b) Which of these four FFTs are purely imaginary?



```
[4.+0.j    1.-2.41j 0.+0.j    1.-0.41j 0.+0.j    1.+0.41j 0.+0.j    1.+2.41j]

[ 2.   +0.j   1.41+0.j   4.   +0.j -1.41+0.j -2.   +0.j -1.41+0.j   4.   +0.j   1.41+0.j]

[0.+0.j    2.-4.83j 0.+0.j    2.-0.83j 0.+0.j    2.+0.83j 0.+0.j    2.+4.83j]

[0.+0.j    0.-4.83j 0.+0.j    0.-0.83j 0.+0.j    0.+0.83j 0.+0.j    0.+4.83j]
```

D.

(c) Can you derive the rules by which a real signal has a purely real FFT or a purely imaginary FFT? (Find the answer in the course notes.)

Answer

Q.1.3 Let $x[n]$ be defined as follows:

$$x[n] = \begin{cases} 1 & for\ n = 0,1,2,3 \\ 0 & otherwise \end{cases}$$

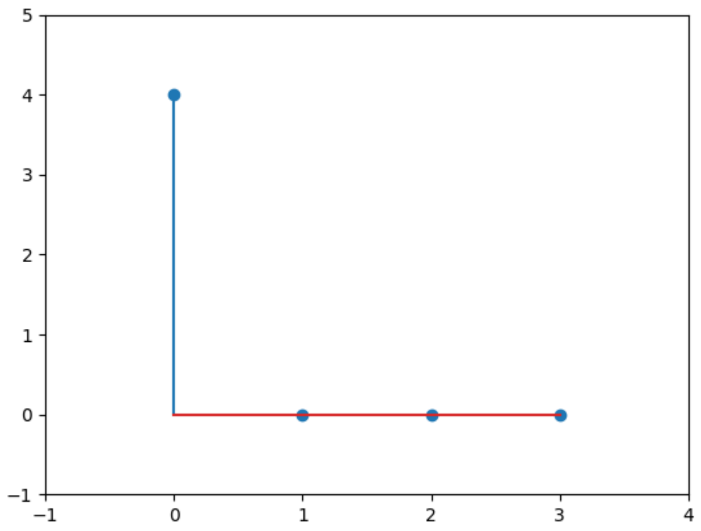
Compute the DFT of $x[n]$.

Procedure:

1. Add a new function `part1_3()` to the program main.py and then call the function in main.
2. Copy the following codes to the function `part1_3()` . The function computes and plots the magnitude of the spectrum of $x[n]$ by using a 4-point DFT (i.e. using only the first 4 ones of $x[n]$). Note how the `pyplot` of `matplotlib` plots the magnitude response of x.

```
def part1_3():
    import numpy as np
    from matplotlib import pyplot as plot
    # np.set_printoptions(precision=2, linewidth=100)

    # Q1.3
    x1 = [1, 1, 1, 1]
    y1 = np.fft.fft(x1)
    magn_x1 = abs(y1)

    plot.stem(magn_x1)
    plot.xlim([-1, 4])
    plot.ylim([-1, 5])
    plot.show()
```

3. Run and program. Copy and paste the plot in the box below.

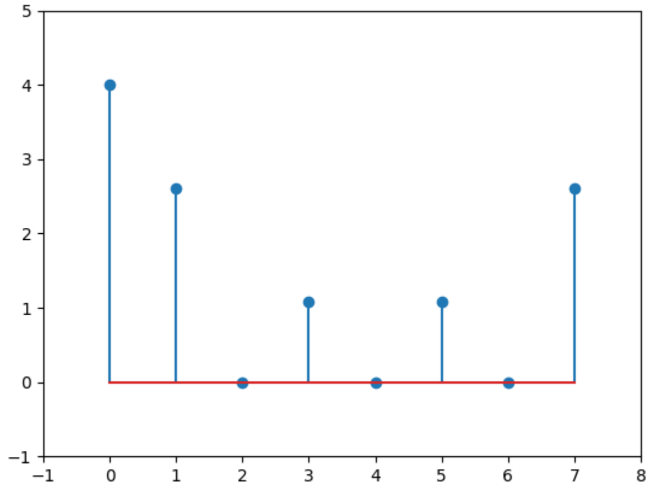


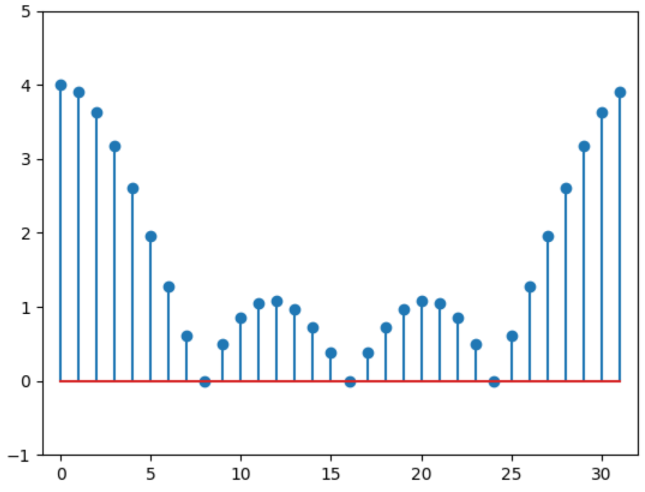
4. With reference to the program above, compute and plot the magnitude of the spectrum of $x[n]$ by using 8, 32, 128 and 1024-point DFTs (by padding appropriate number of zeros). Note how the frequencies are "filled in" as the size of the DFT increases.

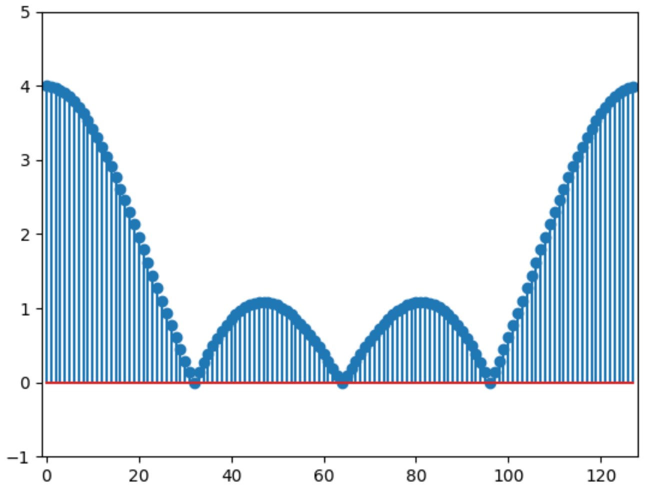
```
5. x1 = [1, 1, 1, 1]

   x2 = np.pad(x1, (0, 4), 'constant')
   x3 = np.pad(x1, (0, 28), 'constant')
```

```
x4 = np.pad(x1, (0, 124), 'constant')
x5 = np.pad(x1, (0, 1020), 'constant')

y1 = np.fft.fft(x1)
y2 = np.fft.fft(x2, 8)
y3 = np.fft.fft(x3, 32)
y4 = np.fft.fft(x4, 128)
y5 = np.fft.fft(x5, 1024)

magn_x1 = abs(y1)
magn_x2 = abs(y2)
magn_x3 = abs(y3)
magn_x4 = abs(y4)
magn_x5 = abs(y5)

plot.stem(magn_x5)
plot.xlim([-1, 1024])  #[-1, x] where x = n of np.fft.fft(xn, n)
plot.ylim([-1, 5])
plot.show()
```

Copy and paste the plots in the box below:







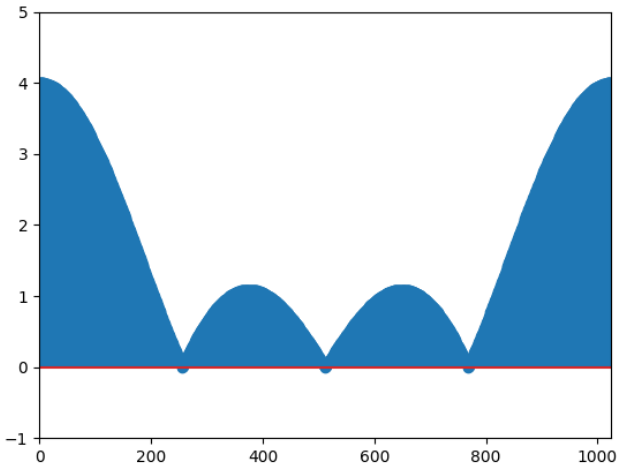


Figure 1,2,3,4 (left to right, up to down): 8, 32, 128, 1024-point DFT of x[n]

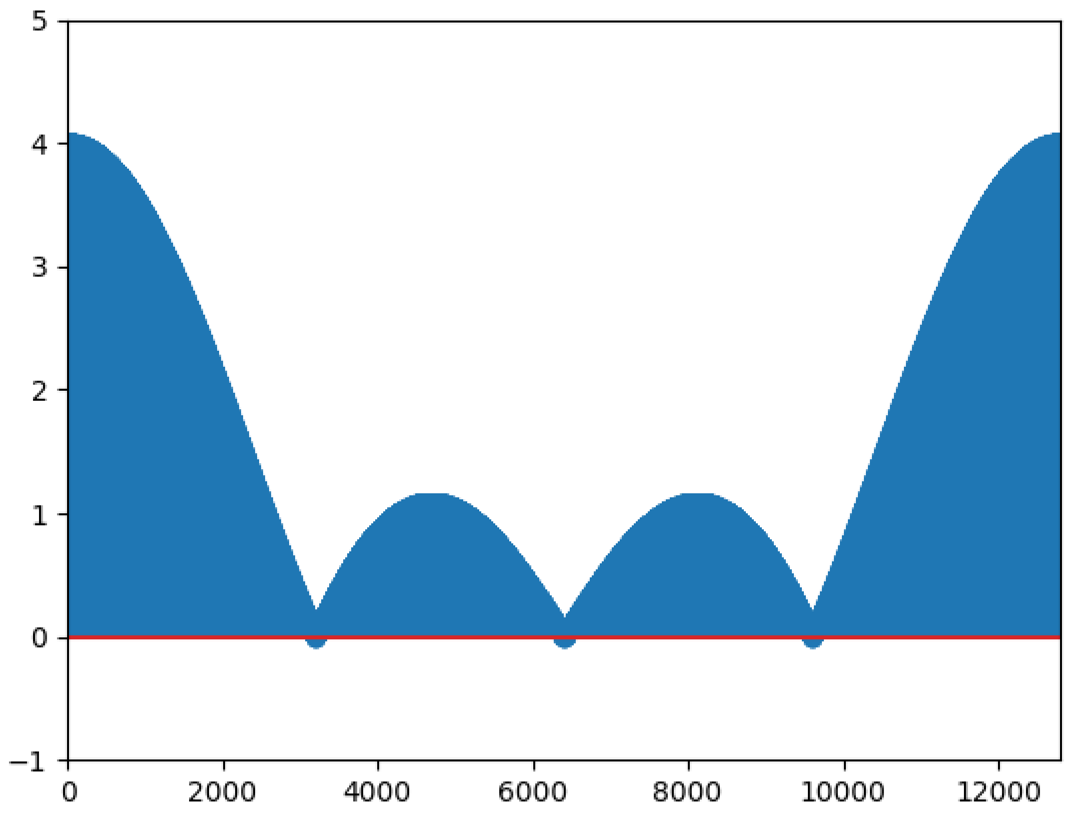
Questions:

(a) Why the spectrum in (a) is so different from the spectrum when 128-point DFT is used?

Assumption: ‘spectrum in (a)’ refers to the spectrum.
Because the x in x-point DFT is basically the resolution of the spectrum. It’s sampled x times and it just so happens that the points on a 4-point DFT is all at 0. Which we can still see occurs in 8,32,128,1024 point DFT.

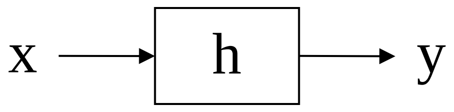
(b) If $X_p(\hat{\omega})$ is the Fourier transform of $x[n]$ and if the sampling frequency is 12,800Hz, find $X_p(\hat{\omega})$ at 900 Hz.

```
: x6 = np.pad(x1, (0, 12796), 'constant')
y6 = np.fft.fft(x6, 12800)
magn_x6 = abs(y6)
print(magn_x6[900])

3.5280971153302696
```



1graph of 12800-point DFT

Part 2. LTI systems and Filtering



Discrete-time linear time-invariant (LTI) systems are usually described by using the linear convolution operator, with input *x*, output *y* and the "impulse response" *h* as follows:

$$y[n] = \sum_{m=0}^{M} x[m]h[n-m] = \sum_{m=0}^{M} x[n-m]h[m].$$

We usually denote the equation above as $y[n] = x[n] * h[n]$. In Python, linear convolution can be computed by the function `convolve` of the `numpy` library. For instance, the statement

```
y = np.convolve(x,h)
```

will compute the convolution of `x` and `h` and give the output `y`. The symbol `np` is an alias pointing to `numpy`.

Q.2.1 Let $x[n]$ and $h[n]$ be defined as follows:

$$x[n] = [1\ \ 2\ \ 3\ \ 4\ \ 5\ \ 6\ \ 5\ \ 4\ \ 3\ \ 2\ \ 1\ \ 1]$$
$$h[n] = [1\ \ 4\ \ 6\ \ 4\ \ 1]$$

Calculate the output of the LTI system *y*[*n*].
Procedure:

1. Add a new function `part2_1()` to the program main.py and then call the function in main.
2. In the function, write a program that compute the linear convolution of $x[n]$ and $h[n]$ by using the command `numpy.convolve`.
3. Show your program and the answer in the box below.

```
def part2_1():
    import numpy as np

    # Q2.1
    x = [1, 2, 3, 4, 5, 6, 5, 4, 3, 2, 1, 1]
    h = [1, 4, 6, 4, 1]
    y = np.convolve(x,h)
    print(y)
```

```
[ 1  6 17 32 48 64 78 84 78 64 48 33 21 12  5  1]
```

Question:
a. What is the length of $y[n]$? How is it related to the length of $x[n]$ and $h[n]$?

16. length(y) = length(x) + length(h) - 1

Q.2.2 It is a very important concept of the LTI system that the linear convolution in time domain is equivalent to the point-by-point multiplication in frequency domain. Let us verify it using x, h, and y as discussed in Q.2.1. Compute the spectrums of $x[n]$, $h[n]$ and $y[n]$ by using a 16-point DFT (hence zeros should be padded to x and h). Verify if $X \circ H$ is equal to Y, where X, H and Y are the length-16 DFTs of x, h and y, respectively. The symbol $\circ$ is point-by-point multiplication.

Procedure:

1. Add a new function `part2_2()` to the program main.py and then call the function in main.
2. In the function, copy the program below to compute the mean squared error between the magnitude of $X \circ H$ and Y using the program below. Note that the mean square difference of two vectors A and B is defined as $\frac{1}{N}\sum_{n=0}^{N-1}(a_n - b_n)^2$, where a_n and b_n are the (n+1)th element of A and B, respectively. Both A and B are assumed to have N elements. In Python, mean squared error can be computed using the function `mean_squared_error` of the `sklearn.metrics` library. To use this library, we need to install it by the following command

```
pip install -U scikit-learn
```

```
def part2_2():
    import numpy as np
    from sklearn.metrics import mean_squared_error as mse

    # Q2.2
    x = [1, 2, 3, 4, 5, 6, 5, 4, 3, 2, 1, 1]
    h = [1, 4, 6, 4, 1]
    y = np.convolve(x, h)

    x01 = np.fft.fft(x, 16)  # Compute the 16-point FFT of x with zero pad
    h01 = np.fft.fft(h, 16)
    y01 = abs(np.multiply(x01, h01))  # point-by-point multiplication
    y02 = np.fft.fft(y)

    diff = mse(abs(y02), abs(y01))
    print(diff)
```

Questions:

(a) Show the mean squared error below. Can it be concluded that $X \circ H = Y$?

4.9134495817953e-28. Yes, accurate to 28 decimal places.

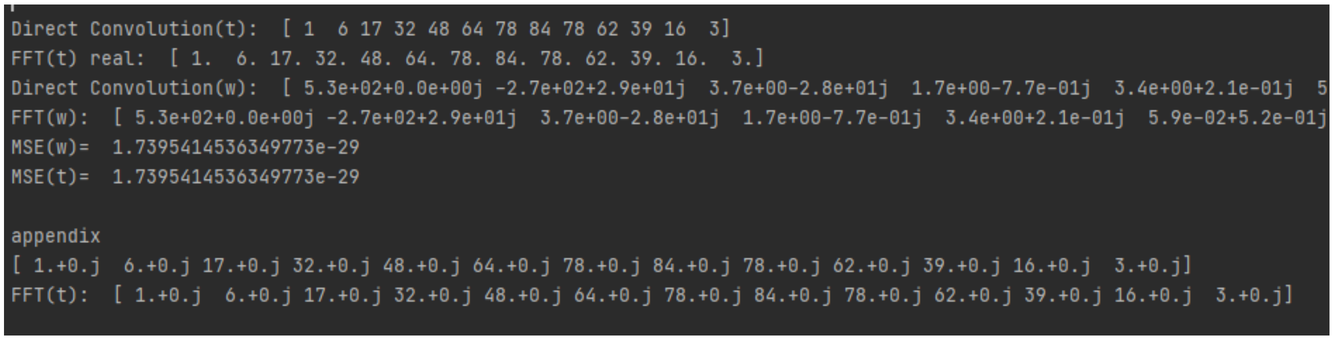
(b) Now the data sequence `x` is changed as follows:

```
x = [1, 2, 3, 4, 5, 6, 5, 4, 3]
```

The convolution output is no longer a power of 2. Thus, we need to pad enough zeros to `x` in order to use the FFT approach. Modify the above program to compute and show the convolution of `x` and `h` using the direct convolution and FFT approach. You may want to use the function `ifft` of `numpy.fft` to do the IFFT. Show your program and program output in the box below

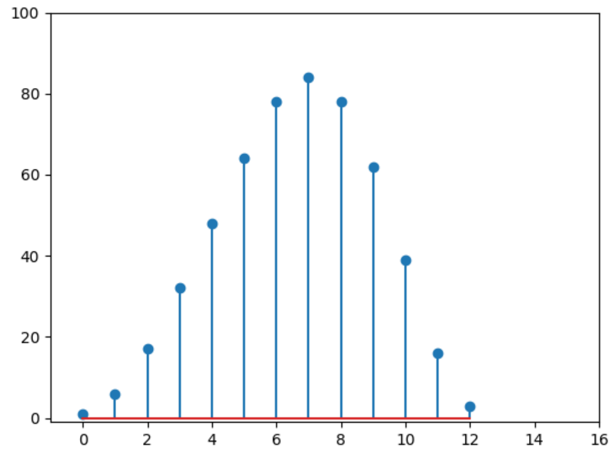
```
def part2_2():
    import numpy as np
    from sklearn.metrics import mean_squared_error as mse
    np.set_printoptions(precision=1, linewidth=300)
    from matplotlib import pyplot as plot
    # Q2.2 Show that linear convolution in time = point-by-point multiplication in
frequency (FFT)
    # x = [1, 2, 3, 4, 5, 6, 5, 4, 3, 2, 1, 1]
    x = [1, 2, 3, 4, 5, 6, 5, 4, 3]
    h = [1, 4, 6, 4, 1]
    x1 = np.pad(x, (0, 4), 'constant')  # pad x=5-1
    h1 = np.pad(h, (0, 8), 'constant')  # pad h=9-1

    # Direct convolution
    y = np.convolve(x, h)
    print('\nDirect Convolution(t): ', y)

    # Point-by-point multiplication: pad, fft, pbp, ifft
    x2 = np.fft.fft(x1)  # fft xp
    h2 = np.fft.fft(h1)  # fft hp
    y1 = np.multiply(x2, h2)  # point-by-point multiplication
    y2 = np.fft.fft(y)  # fft of direct convolution
    y3 = np.fft.ifft(y1)
    y4 = np.fft.ifft(y2)
    print('FFT(t) real: ', np.real(y3))
    print('Direct Convolution(w): ', y2)  #
    print('FFT(w): ', y1)

    diff = mse(abs(y2), abs(y1))
    print('MSE(w)= ', diff)
    diff2 = mse(abs(y3), abs(y4))
    print('MSE(t)= ', diff)

    print('\nappendix')
    print(y4)
    print('FFT(t): ', y3)

    plot.stem(np.real(y3))  # np.real(y3) or y
    plot.xlim([-1, 16])  #[-1, x] where x = n of np.fft.fft(xn, n)
    plot.ylim([-1, 100])
    plot.stem(y)
    plot.show()
```

(c) Are the direct convolution output and FFT approach output exactly the same? If no, what are the differences?



```
Direct Convolution(t):  [ 1  6 17 32 48 64 78 84 78 62 39 16  3]
FFT(t) real:  [ 1.  6. 17. 32. 48. 64. 78. 84. 78. 62. 39. 16.  3.]
Direct Convolution(w):  [ 5.3e+02+0.0e+00j -2.7e+02+2.9e+01j  3.7e+00-2.8e+01j  1.7e+00-7.7e-01j  3.4e+00+2.1e-01j  5
FFT(w):  [ 5.3e+02+0.0e+00j -2.7e+02+2.9e+01j  3.7e+00-2.8e+01j  1.7e+00-7.7e-01j  3.4e+00+2.1e-01j  5.9e-02+5.2e-01j
MSE(w)=  1.7395414536349773e-29
MSE(t)=  1.7395414536349773e-29

appendix
[ 1.+0.j  6.+0.j 17.+0.j 32.+0.j 48.+0.j 64.+0.j 78.+0.j 84.+0.j 78.+0.j 62.+0.j 39.+0.j 16.+0.j  3.+0.j]
FFT(t):  [ 1.+0.j  6.+0.j 17.+0.j 32.+0.j 48.+0.j 64.+0.j 78.+0.j 84.+0.j 78.+0.j 62.+0.j 39.+0.j 16.+0.j  3.+0.j]
```

*2*output



3both convolution outputs in 1 graph. as you can see they're identical

The outputs are exactly the same.

Q.2.3 Now let us see how we can make use of FFT to speed up the computation of linear convolutions.

Procedure:

1. Add a new function `part2_3()` to the program main.py and then call the function in main.
2. Copy the following codes to the function. The codes first generate a random sequence *x* and an FIR filter *h* with random coefficients, and compute their convolution.

```
def part2_3():
    import numpy as np
    from numpy import random
    import time

    x = random.rand(10500)
    h = random.rand(1024)
    tic = time.time()
    y = np.convolve(x, h)
    toc = time.time()
    print(toc-tic)
```

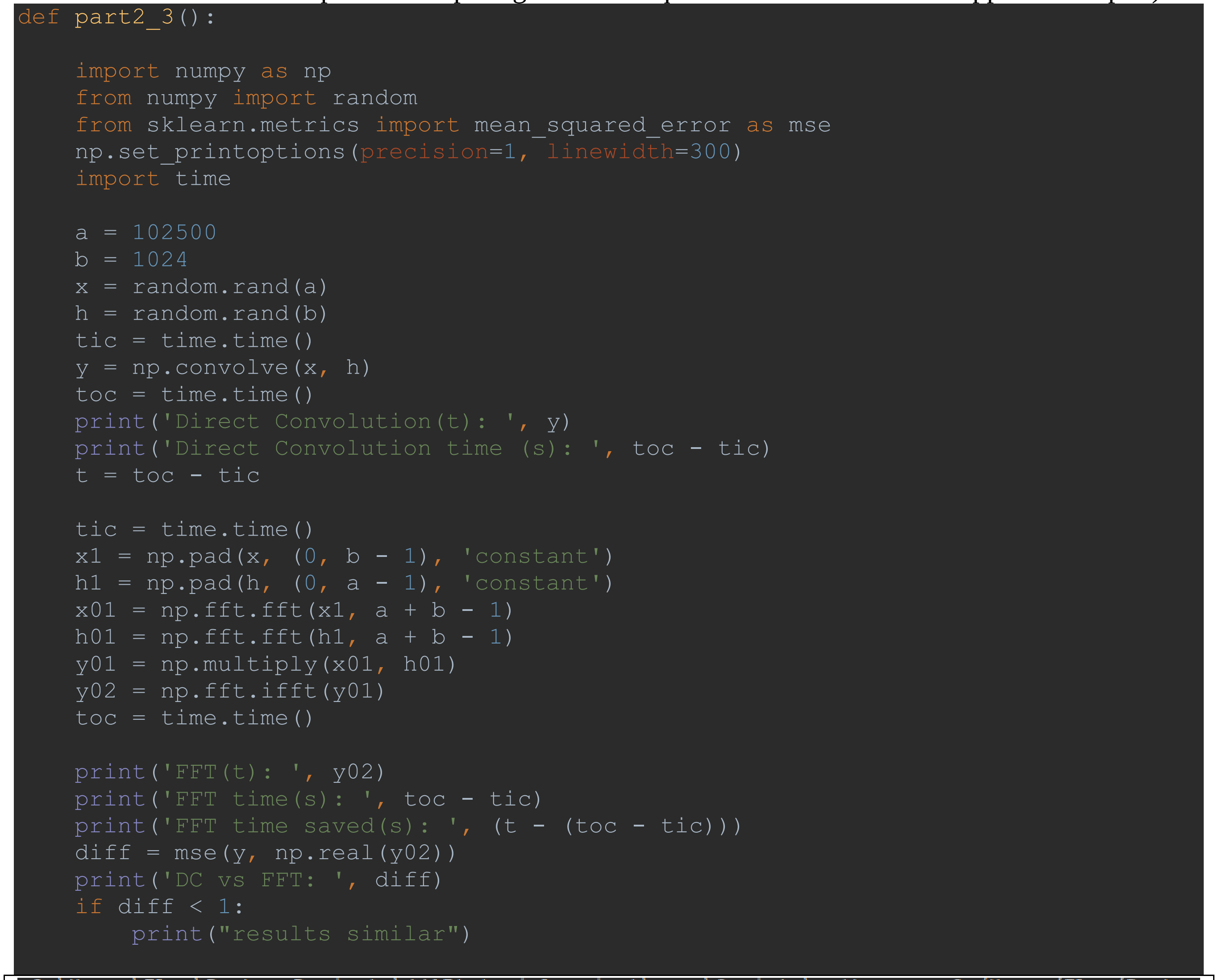
3. Since there are many factors that can affect the computation time of a program, run the above program a few times and calculate the average computation time. Give the value in the box below.

0.007999420, 0.0060178684, 0.00400066375, 0.0039980411, 0.0039968490, 0.0039937496

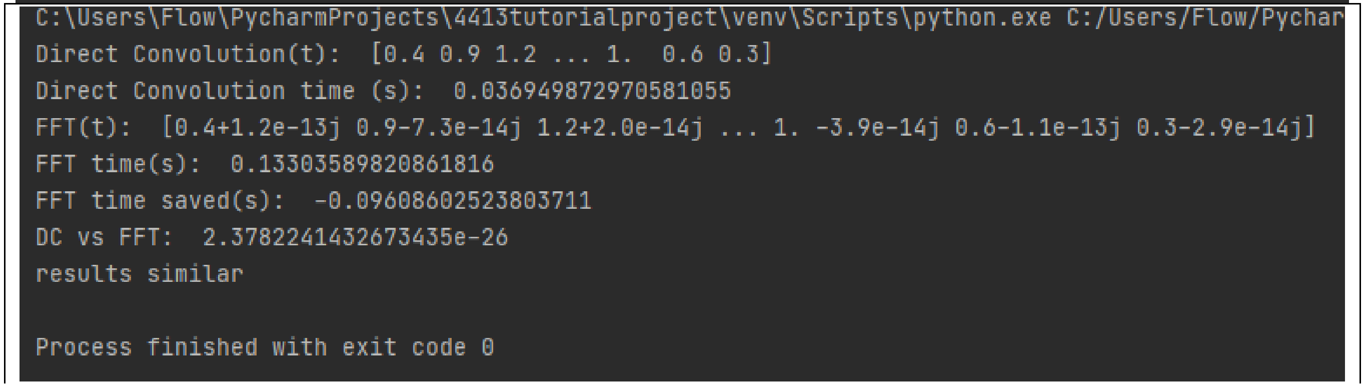
Average = 0.0050010985

4. Modify the above program to obtain the time required to do the same but using the FFT approach. Give your program in the box below. Give also the mean squared error of the outputs given by the direct convolution and FFT approach, as well as the times required for the direct convolution and FFT approach. You will find that the FFT approach does not improve over the direct convolution since many zeros are padded to the input sequence and the filter.

(Note that by default the `ifft` function will give complex valued outputs even if all the imaginary values are zero. The function `mean_squared_error` cannot deal with complex numbers. You should take the real part of the `ifft` output before computing the mean squared error. You may want to use the function `real` of `numpy`. Besides, make sure that you will pad enough zeros to the direct convolution output for computing the mean squared error with the FFT approach output.)



```python
def part2_3():

    import numpy as np
    from numpy import random
    from sklearn.metrics import mean_squared_error as mse
    np.set_printoptions(precision=1, linewidth=300)
    import time

    a = 102500
    b = 1024
    x = random.rand(a)
    h = random.rand(b)
    tic = time.time()
    y = np.convolve(x, h)
    toc = time.time()
    print('Direct Convolution(t): ', y)
    print('Direct Convolution time (s): ', toc - tic)
    t = toc - tic

    tic = time.time()
    x1 = np.pad(x, (0, b - 1), 'constant')
    h1 = np.pad(h, (0, a - 1), 'constant')
    x01 = np.fft.fft(x1, a + b - 1)
    h01 = np.fft.fft(h1, a + b - 1)
    y01 = np.multiply(x01, h01)
    y02 = np.fft.ifft(y01)
    toc = time.time()

    print('FFT(t): ', y02)
    print('FFT time(s): ', toc - tic)
    print('FFT time saved(s): ', (t - (toc - tic)))
    diff = mse(y, np.real(y02))
    print('DC vs FFT: ', diff)
    if diff < 1:
        print("results similar")
```



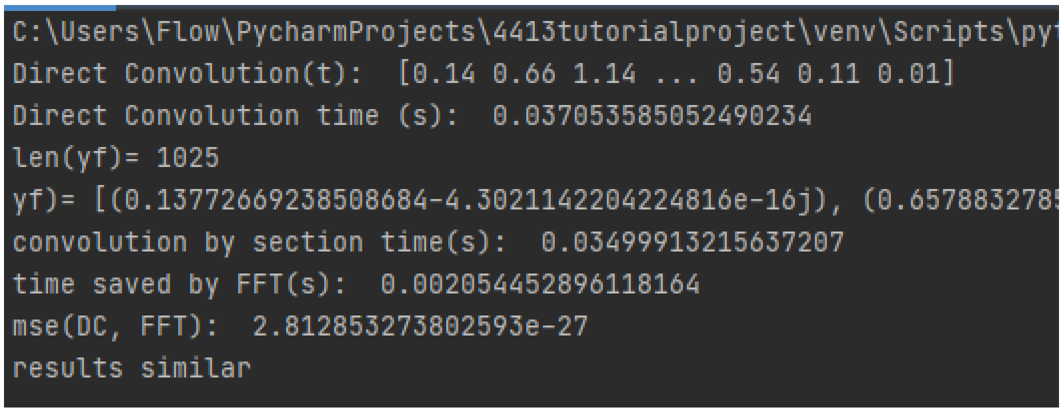
```
C:\Users\Flow\PycharmProjects\4413tutorialproject\venv\Scripts\python.exe C:/Users/Flow/Pychar
Direct Convolution(t):  [0.4 0.9 1.2 ... 1.  0.6 0.3]
Direct Convolution time (s):  0.036949872970581055
FFT(t):  [0.4+1.2e-13j 0.9-7.3e-14j 1.2+2.0e-14j ... 1. -3.9e-14j 0.6-1.1e-13j 0.3-2.9e-14j]
FFT time(s):  0.13303589820861816
FFT time saved(s):  -0.09608602523803711
DC vs FFT:  2.3782241432673435e-26
results similar

Process finished with exit code 0
```

Q.2.4 Now we use the overlap and add method to speed up the operation.

Procedure:

1. Add a new function `part2_4()` to the program main.py and then call the function in main.
2. In the function, write a program that implements the linear convolution in Q.2.3 using the convolution by section method. Speed up the convolution computation by using the FFT approach. You may want to follow the procedure below:
   - Divide the sequence `x` into 100 segments, each with 1025 data.
   - Compute the DFT of `h` (i.e. `H`) by using a 2048-point FFT
   - For each segment x_r, compute its DFT (i.e. X_r) by using a 2048-point FFT. Then dot-multiply with `H` to generate Y_r.
   - Compute the inverse FFT of Y_r to obtain y_r.
   - Sum all y_r to get the final convolution result (note that each y_r starts at different point of the final sequence).
3. Give your program in the box below. Give also the mean squared error of the outputs given by the direct convolution and FFT approach, as well as the times required for the direct convolution and FFT approach. Since there are many factors affecting the timing measurement, you should run the program a few times and take the average as the final result. You will find the overlap and add approach with FFT has a lower computational complexity compared to the direct convolution approach.

```
def part2_4clean():

    # q2.4 overlap and add method vs direct convolution
    import numpy as np
    from numpy import random
    from sklearn.metrics import mean_squared_error as mse
    np.set_printoptions(precision=2, linewidth=500)
    import time

    a = 102500
    b = 1024
    x = random.rand(a)
    h = random.rand(b)

    # direct convolution
    tic = time.time()
    y = np.convolve(x, h)
    toc = time.time()
    print('Direct Convolution(t): ', y)
    print('Direct Convolution time (s): ', toc - tic)
    t = toc - tic

    # convolution by section
    tic = time.time()
    xspl = np.split(x, 100)
    H = np.fft.fft(h, 2048)
    X = np.fft.fft(xspl, 2048)
    Yr = np.multiply(X, H)
    yr = np.fft.ifft(Yr)
    yf = []
    yf.extend(yr[0][0:1025])
    print('len(yf)=', len(yf))
    print('yf)=', yf)
    for i in range(0, 99):
        yf.extend(yr[i][1025:2048] + yr[i + 1][0:1023])
        yf.extend(yr[i + 1][1023:1025])
    yf.extend(yr[99][1025:])
    toc = time.time()
    diff = mse(np.real(y), np.real(yf))
    print('convolution by section time(s): ', toc - tic)
    print('time saved by FFT(s): ', t - toc + tic)
    print('mse(DC, FFT): ', diff)
    if diff < 1:
        print("results similar")
```



```
C:\Users\Flow\PycharmProjects\4413tutorialproject\venv\Scripts\py
Direct Convolution(t):  [0.14 0.66 1.14 ... 0.54 0.11 0.01]
Direct Convolution time (s):  0.037053585052490234
len(yf)= 1025
yf)= [(0.13772669238508684-4.3021142204224816e-16j), (0.6578832785
convolution by section time(s):  0.03499913215637207
time saved by FFT(s):  0.002054452896118164
mse(DC, FFT):  2.812853273802593e-27
results similar
```

4. **<u>Show to your tutor how your program is executed and the execution times of different approaches.</u>**

The End

References

[1] Chapter 2 and 3, Course notes, EIE4413.
[2] S.K. Mitra, Digital Signal Processing, 3rd Edition, McGraw-Hill Education (Asia), 2009.
[3] https://www.w3schools.com/python/
[4] https://www.tutorialspoint.com/python/index.htm