Bu-Ali Sina University

Report

Singal and System Project

Mehran Shahidi

supervised by
Dr. Mahdi Abasi

February 15, 2020

Summary

This is a report for Signal and System course project. In this project, I have used Python for implementing some basic concepts in signal and system. For understanding, this project, some basic knowledge about Signal and System required. Basic Python and MatLab programming knowledge can be helpful. For more information and source codes please visit my GitHub repo for this project.

https://github.com/m3hransh/Signal_project

Contents

1 introduction

In this project, I have used two famous libraries in Python **Numpy** and **Matplotlib**. In this section I am going to give some summary about them.

1.1 Numpy

NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

1.2 Matplotlib

Matplotlib is a plotting library for the Python programming language and its numerical mathematics extension NumPy. It provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits like Tkinter, wxPython, Qt, or GTK+.

2 Plotting Signals

2.1 Q1.1

In this section, I show you how to implement **unit step** and **ramp** function and how to plot them with the different sampling rates with the use of **Numpy** and **Matplotlib** in python.

2.1.1 Unit Step

Unit step is a signal with magnitude one for time greater than zero . We can assume it as a dc signal which got switched on at time equal to zero. this is the mathematical definition of unit step:

$$x(t) = \begin{cases} 0 & : t < 0 \\ 1 & : t \geq 0 \end{cases} \tag{1}$$

The unit step function as it is shown in the **Listing 1**, takes one **Numpy** array or simple list as an input (samples of the time) and then return a list of output. this is list comprehension in Python that iterates through time samples and for each time samples if the sample is less than 0 it returns 0 and otherwise returns 1 (this is the exact definition of the unit step function).

```python
def unit(x):
    # To support scalar values
    if isinstance(x, Iterable):
        return np.array([int(i>=0) for i in x ])
    else:
        return int(x>=0)
```

Listing 1: **unit step** Signal

for the next part, we are using the function that was defined to plot the diagrams for sampling rates of $F_s = 100$ and $F_s = 1000$ (**Listing 2**). the code is straightforward. first, it makes two grid in a row, and gets two axes and drawing the unit function with $F_s = 100$ in the first axis and $F_s = 1000$ in the second one. plots is shown in **Figure 1**.

```python
# create a figure with two axes in a row
fig, (ax1,ax2) = plt.subplots(2,1,figsize=(11,5))
#sampling with rate of 10
t1= np.linspace(-5,5,100)
u = unit(t1)
ax1.plot(t1,u,'.-')
ax1.set_xlabel('t')
ax1.set_ylabel(r'$x(t)$')
```

```python
ax1.set_title(r'$u(t)$ with sampling rate of $F_s=10$')

#sampling with rate of 100
t2 = np.linspace(-5,5,1000)
u2 = unit(t2)
ax2.plot(t2,u2,'.-')
ax2.set_xlabel('t')
ax2.set_ylabel(r'$x(t)$')
ax2.set_title(r'$u(t)$ with sampling rate of $F_s=100$')
plt.subplots_adjust(hspace=0.5)
fig.savefig('doc/images/Q1-1-unit.pgf')
fig.show()
```

Listing 2: Plotting **unit step** with two sampling rates

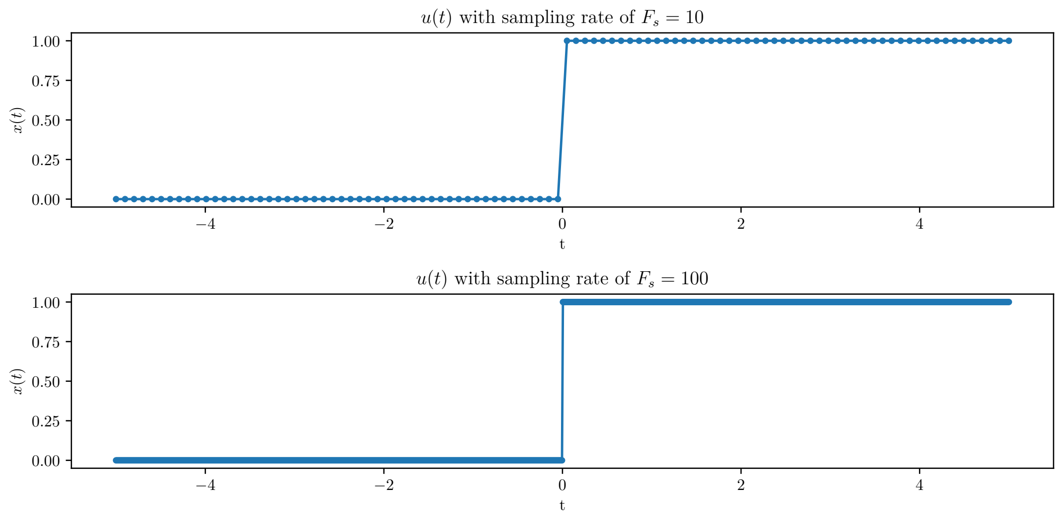

Figure 1: Graph result of the **Listing 2**

2.1.2 Ramp

The ramp signal definition is shown in expression 2. For the negative values, it returns zero and otherwise returns the value of the input. The implementation is also straightforward as the definition. You can see the implementation in python in **Listing 3**. The function takes one input as a simple list or **Numpy** array and returns the output list in the same structure that has explained for the unit signal implementation.

$$x(t) = \begin{cases} x & : x \geq 0 \\ 0 & : x < 0 \end{cases} \tag{2}$$

```python
def ramp(x):
    if isinstance(x,Iterable):
        return np.array([0 if i<0 else i for i in x])
    else:
        return 0 if x<0 else x
```

Listing 3: **Ramp** Signal implementation

For the next part, we are using the function that was defined to plot graphs for sampling rates of $F_s = 100$ and $F_s = 1000$ (**Listing 4**). the code is straightforward. first, it makes two grid in a row, and gets two axes and drawing the unit function with $F_s = 100$ in the first axis and $F_s = 1000$ in the second one. plots is shown in **Figure 2**.

```python
# create a figure with two axes in a row
fig, (ax1,ax2) = plt.subplots(2,1,figsize=(11,5))

#sampling with rate of 10
t1= np.linspace(-5,5,100)
u = ramp(t1)
ax1.plot(t1,u,'.-')
ax1.set_xlabel('t')
ax1.set_ylabel(r'$x(t)$')
ax1.set_title(r'$r(t)$ with sampling rate of $F_s=10$')

#samping with rate of 100
t2 = np.linspace(-5,5,1000)
u2 = ramp(t2)
ax2.plot(t2,u2,'.-')
ax2.set_xlabel('t')
ax2.set_ylabel(r'$x(t)$')
ax2.set_title(r'$r(t)$ with sampling rate of $F_s=100$')
plt.subplots_adjust(hspace=0.5)
plt.savefig('doc/images/Q1-1-ramp.pgf')
fig.show()
```

Listing 4: Plotting **ramp** with two sampling rates

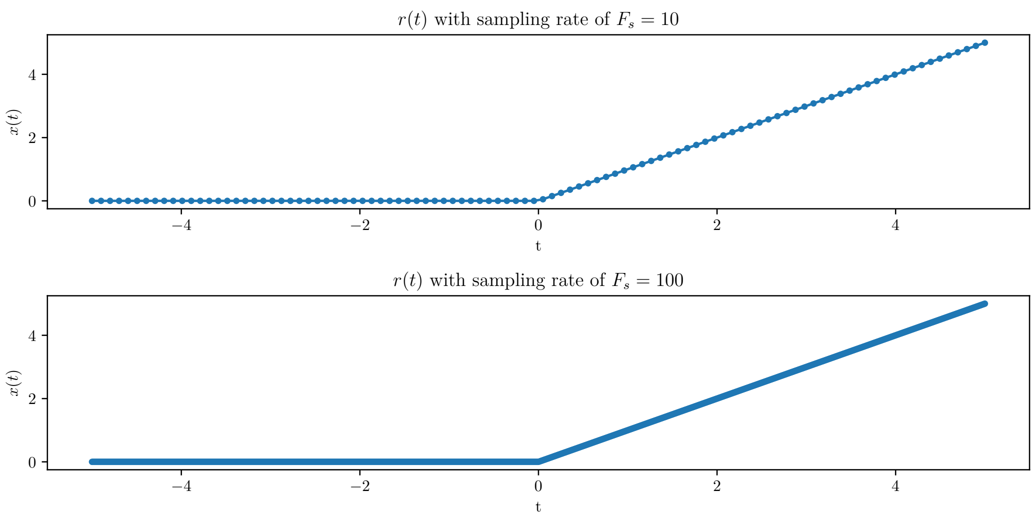

Figure 2: Graph result of the **Listing 4**

2.2 Q1.2

In this part, we are going to plot an example graph using what we have defined before. The example has shown in the **Figure 3**.

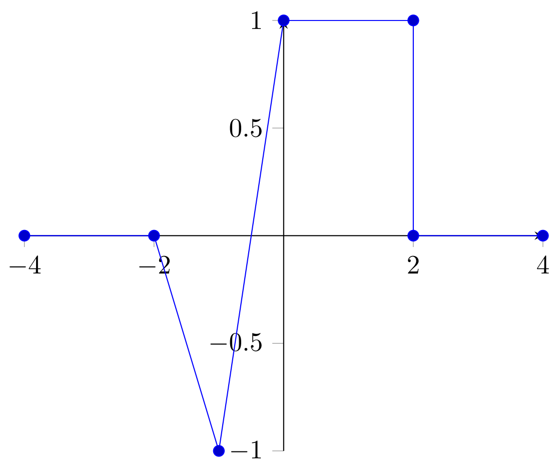

Figure 3: example graph of the siganl $x(t)$

So now, we define our graph using ramp and unit signal and plot them using that definition in Python. We split our graph into three parts. our first part is from $-\infty$ to -1 that is equal to $-r(t+2)*u(-t-1)$. The second part is from -1 to 0 that is equal to $(r(2*t+2)-1)*(u(t+1)-u(-t))$. The third part is from 0 to ∞ that is equal to $u(t)-u(t-2)$. Our final definition for our signal is :

$$x(t) = -r(t+2)*u(-t-1)+(r(2*t+2)-1)*(u(t+1)-u(-t))+u(t)-u(t-2) \tag{3}$$

The implementation of the signal in Python has been shown in the **Listing 5**. First, each part calculated separately, and then the main signal is plotted as the sum of each part. The result of code also has been shown in the **Figure 4**.

```python
# create grid for the figure
gridsize = (3, 3)
fig = plt.figure(figsize=(16,11))
# The main siganl is on the right with two column to make it bigger
ax = plt.subplot2grid(gridsize, (0,1), rowspan=3,colspan=2)
ax1 = plt.subplot2grid(gridsize, (0,0))
ax2 = plt.subplot2grid(gridsize,(1,0))
ax3 = plt.subplot2grid(gridsize, (2,0))

# Sampling with the rate 8/1000
t= np.linspace(-4,4,1000)
# Parts of function
x1 = -ramp(t+2) *unit(-t-1)
x2 = (ramp(2*t+2)-1) * (unit(t+1)-unit(t))
x3 = unit(t) - unit(t-2)
x = x1+x2+x3

# Plotting first part
ax1.plot(t,x1)
ax1.set_title(r'$x_1(t)$')
ax1.set_xticks(np.arange(-4,4,step=1))

# Plotting second part
ax2.plot(t,x2)
ax2.set_title(r'$x_2(t)$')
ax2.set_xticks(np.arange(-4,4,step=1))

# Plotting third part
ax3.plot(t,x3)
ax3.set_title(r'$x_3(t)$')
ax3.set_xticks(np.arange(-4,4,step=1))

# Plotting the full signal
ax.plot(t,x)
ax.set_xlabel('t')
ax.set_yticks(np.arange(-1,2,step=1))
ax.set_title(r'$x(t) = x_1(t) + x_2(t) + x_3(t)$')
ax.set_xticks(np.arange(-4,4,step=1))

plt.savefig('doc/images/Q1-2-ex1.pgf')
fig.show()
```

Listing 5: Plotting signal $x(t)$ and its parts

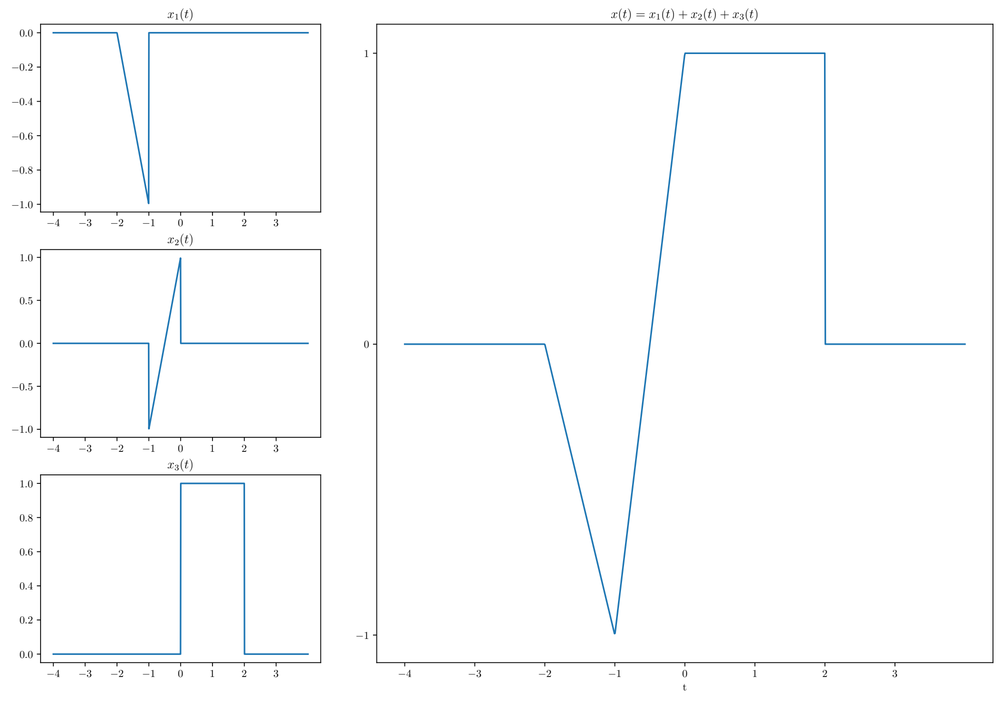

Figure 4: Graph result of the **Listing 5**

2.3 Q1.3

In this part, we are going to plot the signal $y(t)$ that has been shown in the **Figure 5**. we can find the signal $y(t)$ concerning $x(t)$ that has defined in the previous section. We can find $y(t)$ by doing some transformation on $x(t)$. One way to see it is to first shift $x(t)$ 2 unit to the left and then compress it with the scale number of 2.

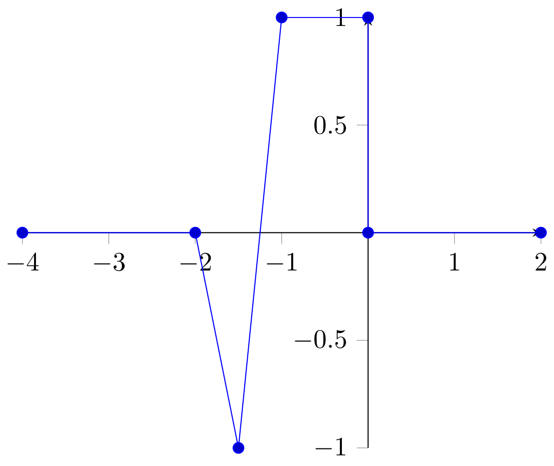

Figure 5: example graph of the siganl $y(t)$

So we can easily find $y(t)$ in respect to $x(t)$ just that is $y(t) = x(2t+2)$. For implementation in Python we do the same. after defining function $x(t)$ we find $y(t)$ in a same manner. The code has been shown in **Listing 6**. And the corospnding graph result has been shown in **Figure 6**.

```python
fig, ax = plt.subplots(1,1)

t = np.linspace(-4,2,1000)
# Definition of the x(t) signal as a lambda function
x = lambda t : -ramp(t+2) *unit(-t-1) +\
               (ramp(2*t+2)-1) * (unit(t+1)-unit(t)) +\
               unit(t) - unit(t-2)
# Shifting x(t) to the left and compress it
y = lambda t:x(2*t+2)
ax.plot(t,y(t))
ax.set_xlabel('t')
ax.set_ylabel(r'$y(t)$')
ax.set_yticks(np.arange(-1,2,step=1))

plt.savefig('doc/images/Q1-3-ex2.pgf')
fig.show()
```

Listing 6: Plotting signal $y(t)$

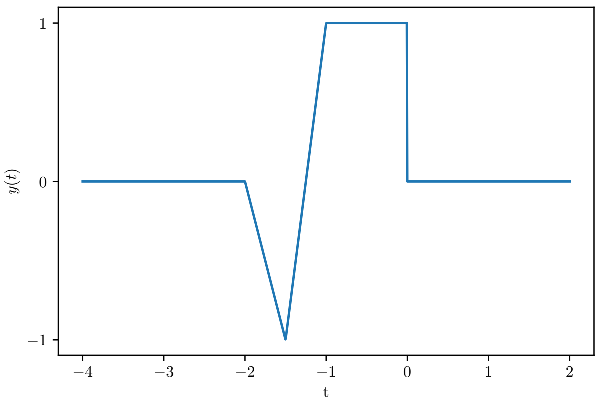

Figure 6: Graph result of the **Listing 6**

2.4 Q1.4

In this part, we are going to plot another signal, $y_2(t)$ that has been shown in the **Figure 7**. The process is like the previous section. We can find $y_2(t)$ by first shifting the $x(t)$ to the left and then mirror it with respect to y axis.

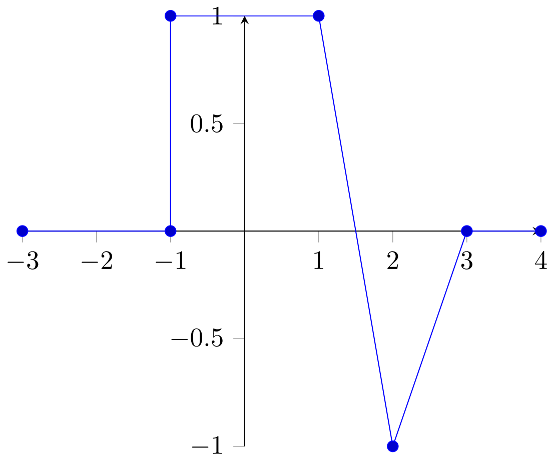

Figure 7: example graph of the siganl $y_2(t)$

So we can easily find $y_2(t)$ with respect to $x(t)$ that is $y_2(t) = x(-t+1)$. For implementation in Python we do the same. after defining function $x(t)$ we find $y_2(t)$ in a same manner. The code has been shown in **Listing 7**. And the corospnding graph result has been shown in **Figure 8**.

```python
fig, ax = plt.subplots(1,1)

t = np.linspace(-3,4,1000)
# Definition of the x(t) signal as a lambda function
x = lambda t : -ramp(t+2) *unit(-t-1) +\
               (ramp(2*t+2)-1) * (unit(t+1)-unit(t)) +\
               unit(t) - unit(t-2)
# Shifting x(t) to the left and mirror it
y2 = lambda t:x(-t+1)
ax.plot(t,y2(t))
ax.set_xlabel('t')
ax.set_ylabel(r'$y_2(t)$')
ax.set_yticks(np.arange(-1,2,step=1))

plt.savefig('doc/images/Q1-4-ex3.pgf')
fig.show()
```

Listing 7: Plotting signal $y_2(t)$

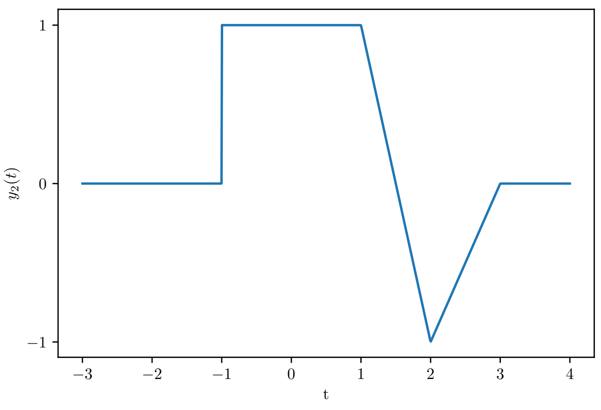

Figure 8: Graph result of the **Listing 7**

3 Signal Analysis

In this section, we will define some simple systems with one input signal and one output signal. After defining them, we will evaluate its response to the signal $x(t)$ that defined in the previous section.

$$x_e[n] = \frac{x[n]-x[-n]}{2} \qquad x_e[n] = \frac{x[n]+x[-n]}{2}$$

$$x_t[n] = \begin{cases} x[n] & : n < 0 \\ 0 & : o.w \end{cases} \qquad x_r[n] = \begin{cases} x[n] & : n \geq 0 \\ 0 & : o.w \end{cases}$$

For implementation in Python, first, we define functions for each expression above with one input function that returns a corresponding response function. The code has been shown in the **Listing 8**. Take note that each function returns a one-line lambda function. After that in the following lines, each of them invokes using signal $x(t)$ as an input function that returns the corresponding response function as an output.

```python

# Defintion of systems that take a signal x as an input and return siganl f
fx_o = lambda x: lambda n:(x(n) - x(-n))/2
fx_e = lambda x: lambda n: (x(n) + x(-n))/2
fx_t = lambda x: lambda n: np.array([x(np.array([i]))[0] if i<0 else 0 for i in n])
fx_r = lambda x: lambda n: np.array([x(np.array([i]))[0] if i>=0 else 0 for i in n])

# Call them on the signal x(t)
x_o = fx_o(x)
x_e = fx_e(x)
x_t = fx_t(x)
x_r = fx_r(x)
```

Listing 8: Definition of systems in Python

3.1 Q2.1

Now lets, plot them with **Pyplot**. The code is in **Listing 9**. In the first line, a grid with two rows and two columns has defined. And each signal has plotted in each cell of the grid (result graphs in **Figure 9**).

```python
# Sampling with rate of 100
n = np.linspace(-4,4,800)

# Making figure with 2 rows and 2 columns
gridsize = (2, 2)
fig = plt.figure(figsize=(16,11))
```

```
ax1 = plt.subplot2grid(gridsize, (0,0))
ax2 = plt.subplot2grid(gridsize, (0,1))
ax3 = plt.subplot2grid(gridsize,(1,0))
ax4 = plt.subplot2grid(gridsize, (1,1))

ax1.plot(n, x_o(n))
ax1.set_title(r'$x_o(n)$')
ax1.set_xticks(np.arange(-4,5,step=1))
ax1.set_yticks(np.arange(-1,1.5,step=.5))

ax2.plot(n, x_e(n))
ax2.set_title(r'$x_e(n)$')
ax2.set_xticks(np.arange(-4,5,step=1))
ax2.set_yticks(np.arange(-1,1.5,step=.5))

ax3.plot(n, x_t(n))
ax3.set_title(r'$x_t(n)$')
ax3.set_xticks(np.arange(-4,5,step=1))
ax3.set_yticks(np.arange(-1,1.5,step=.5))

ax4.plot(n, x_r(n))
ax4.set_title(r'$x_r(n)$')
ax4.set_xticks(np.arange(-4,5,step=1))
ax4.set_yticks(np.arange(-1,1.5,step=.5))

fig.show()
```

Listing 9: plotting response function of each system to $x(t)$

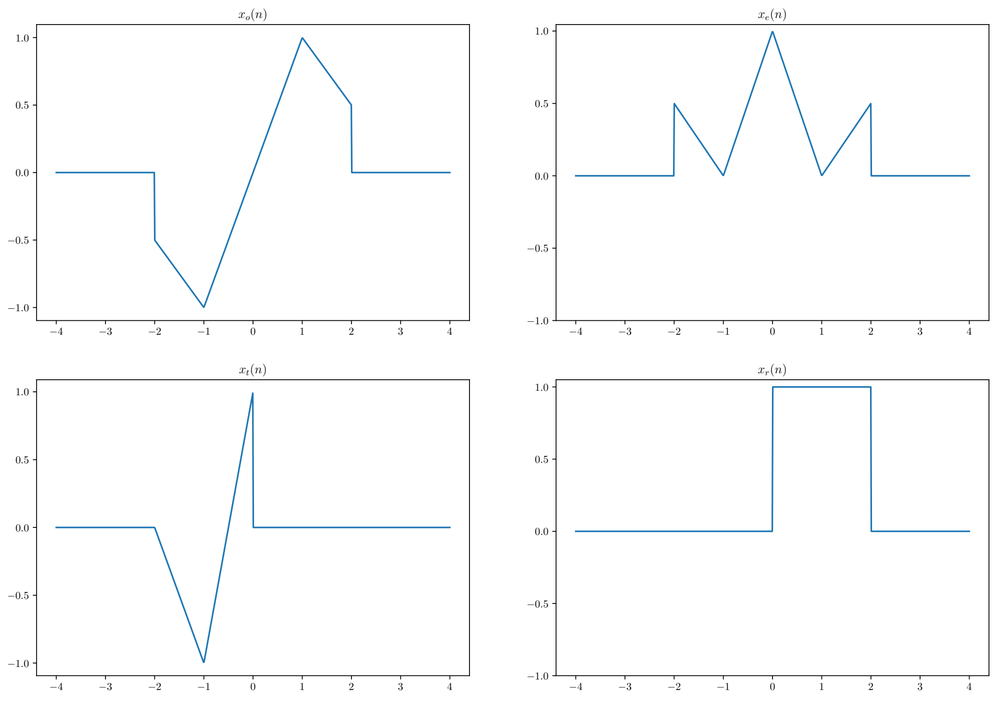

Figure 9: Result graph of the **Listing 9**

3.2 Q2.2

Let, see if we can rebuild signal $x(t)$ from signals $x_e(t)$ and $x_r(t)$. We know that $x_e(t)$ is even part of the signal $x(t)$ so if we find the odd part of the signal from $x_r(t)$, we can regenerate $x(t)$. if we subtract signal

$x_e(t)$ from $x_r(t)$ when n>=0 we will find the odd part of the signal $x(t)$ and we can also find negative values of odd signals that is negetive mirror of its positive values. So siganl x(t) is :

$$x(t) = \begin{cases} x_r(t) & : t \geq 0 \\ x_e(t) - x_r(-t) + x_e(-t) & : o.w \end{cases} \tag{4}$$

The implementation has been shown in **Listing 10**. The generated $x(t)$ has defined using the definition above and then, the regenerated and original $x(t)$ have plotted in a figure with two axes in a row.

```python

fx_e = lambda x: lambda n: (x(n) + x(-n))/2
fx_r = lambda x: lambda n: np.array([x(np.array([i]))[0] if i>=0 else 0 for i in n])
x_e = fx_e(x)
x_r = fx_r(x)

# x(t) for t>=0
nx = lambda t: 2*x_e(t) - x_r(-t)
# The function concatenate the negetive part with positive
xg = lambda t: np.concatenate((nx(np.array([i for i in t if i<0])),x_r(np.array([i for i
    in t if i>=0]))))

# Sampling with rate of 100
t = np.linspace(-4,4,800)
fig, (ax1,ax2) = plt.subplots(1,2,figsize=(16,6))

# Plotting original x(t)
ax1.plot(t, x(t))
ax1.set_title('original $x(t)$')
ax1.set_xticks(np.arange(-4,5,step=1))
ax1.set_yticks(np.arange(-1,1.5,step=.5))

# Plotting generated signal
ax2.plot(t, xg(t))
ax2.set_title('generated $x(t)$ using signals $x_r(t)$ and $x_e(t)$')
ax2.set_xticks(np.arange(-4,5,step=1))
ax2.set_yticks(np.arange(-1,1.5,step=.5))

fig.show()
```

Listing 10: Regneerating signal $x(t)$ from $x_e(t)$ and $x_r(t)$

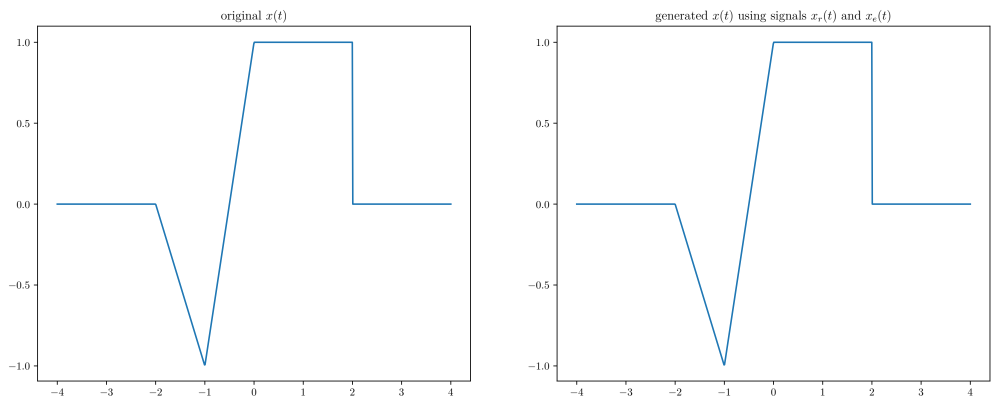

Figure 10: Result graph of the **Listing 10**

3.3 Q2.3

Now, let's see if we can also regenerate signal $x(t)$ from signals $x_o(t)$ and $x_t(t)$. We can use the same logic, we have the $x(t)$ values for $t < 0$ and we can find the even part of the $x(t)$ by subtracting $x_o(t)$

from $x_t(t)$ and using the symmetry property of the even functions to find even part for positive values of t. But the only problem that still remains is that we don't know the value of $x(t)$ for $t = 0$ and its value is not in $x_o(t)$ nor $x_t(t)$. So it is impossible to regenrate the exact $x(t)$.

4 Systems

In this section, a system has been defined. The definition has been shown below:

$$y(t) = \int_{-\infty}^{+\infty} e^{u-t} x(u-2) du \tag{5}$$

4.1 Q3.1

Now, we are going to show if the system has a superposition property. To prove that we will show that $S\{a_1x_1(t) + a_2x_2(t)\}$ and $S\{a_1x_1(t)\} + S\{a_2x_2(t)\}$ give us same result. ($x_1(t)$ and $x_2(t)$ has been defined below)

$$x_1(t) = u(t) - u(t-2) \tag{6}$$
$$x_2(t) = u(t) - u(t-3) \tag{7}$$

For implementing the system in Python we will do the same as the before. First, defining a function that has an input function and returns an output function with the use of lambda expression in Python. For the Integrating part of it, I have used **SciPy**. For importing the Integral, I have used the line below at the top of my code.

```python
import scipy.integrate as integrate
```

The Implementation is straightforward (**Listing 11**). I have chosen a_1 as 3 and a_2 as 2 and after defining the x_1 and x_2, $S\{a_1x_1(t) + a_2x_2(t)\}$ and $S\{a_1x_1(t)\} + S\{a_2x_2(t)\}$ has been defined. Then they have plotted in a row and the result has been shown in the **Figure 11**.

```python
# Defining the y(t) systems that takes signal as an input
y = lambda f: lambda t:np.array([integrate.quad(lambda u:f(u-2)*np.exp(u-i),-10,10)[0] for
     i in t])
x1 = lambda t: unit(t) - unit(t-2)
x2 = lambda t: unit(t) - unit(t-2)
a1=3
a2=2
# S{a1x1(t) + a2x2(t)}
y1 = y(lambda t: a1*x1(t)+a2*x2(t))
# S{a1x1(t)} + S{a2x2(t)}
y2 = lambda t1:y(lambda t: a1*x1(t))(t1)+ y(lambda t : a2*x1(t))(t1)
# Sampling with rate of 100
n = np.linspace(-5,5,1000)
# Making the figure with 2 axes in a row
fig, (ax1,ax2) = plt.subplots(1,2,figsize=(12,6))

ax1.plot(n, y1(n))
ax2.plot(n, y2(n))
fig.show()
```

Listing 11: plotting response function of the system to $S\{a_1x_1(t) + a_2x_2(t)\}$ and $S\{a_1x_1(t)\} + S\{a_2x_2(t)\}$

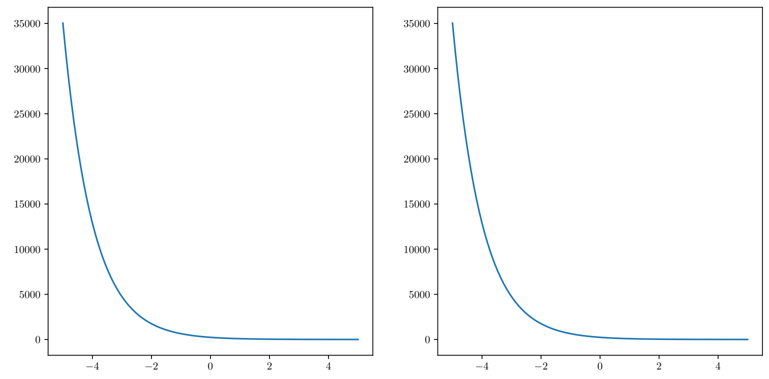

Figure 11: Result graph of the **Listing 11**

4.2 Q3.2

In the previous section, we have shown by example that the system of equation 5 is Linear. In this section, we will show by example if the system is also **time-invariant**. Imagine $y_1(t)$ be the response function of $x_1(t)$ in the equation 6 and $y_2(t)$ be the response function of system to $x_1(t-3)$. We will show that both are the same and this is in compliance with the time-invariant property of the system.

Implementation In python is also simple as it's shown in the **Listing 12**. First the $y_1(t)$ and the $y_2(t)$ signal has been defined using y system and x_1 signal that has been defined in **Listing 11** in the previous section. The result also has been shown in the **Figure 12**. As you can see, both of them are the same.

```python
# y1(t)
y1 = y(x1)
# y2(t)
y2 = y(lambda t: x1(t-3))

n = np.linspace(-5,5,1000)

fig, (ax1,ax2) = plt.subplots(1,2,figsize=(12,6))

ax1.plot(n, y1(n-3))
ax2.plot(n, y2(n))
fig.show()
```

Listing 12: plotting response function of the system to $S\{a_1x_1(t) + a_2x_2(t)\}$ and $S\{a_1x_1(t)\} + S\{a_2x_2(t)\}$

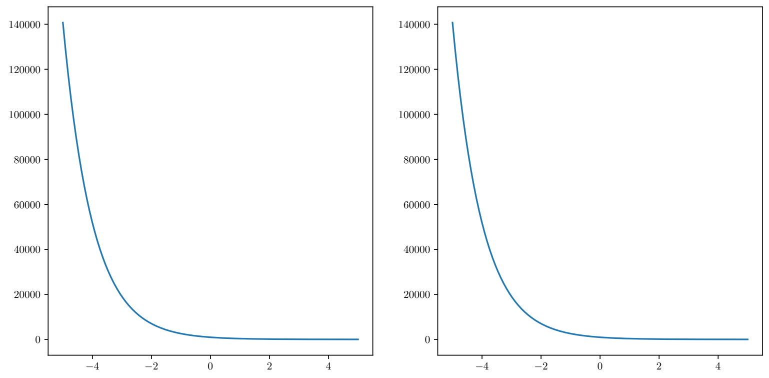

Figure 12: Result graph of the **Listing 12**

4.3 Q3.3

In two previous sections, we have shown the **linearity** and **time-invariant** property of the system in equation 5. Although we haven't proven those properties, we have shown instances that comply with those. So we can say that the system is **LTI**.

5 Signal Energy

In this section, we will define a general function for calculating the signal energy. the total energy in a discrete-time signal $x[n]$ over the time interval $n_1 \leq n \leq n_2$ is defined as:

$$\sum_{n=n_1}^{n_2} |x[n]|^2 \tag{8}$$

5.1 Q4.1

The implementation in python is as simple as this one line code:

```python
dt_energy = lambda x, n: sum([np.abs(x(i))**2 for i in n])
```

Now let's amend the function to calculate the continuous-time signal energy. The total energy over the time interval $t_1 \leq t \leq t_2$ in a continuous-time signal $x(t)$ is defined as

$$\int_{t_1}^{t_2} |x(t)|^2 dt \tag{9}$$

But in practice, we can't precisely calculate the integral, So instead, we can use Riemann sum as follow

$$\lim_{n\to\infty} \sum_{i=1}^{n} |x(t_i)|^2 \Delta t \tag{10}$$

that n is the number of samples and Δt is the distance btween samples.

So for implementation, the only change that we should do in comparison to the discrete-time signals is to multiply each sample to the distance between them. The code is like this:

```python
# d is the distance between samples
ct_energy = lambda x, n, d: sum([np.abs(x(i))**2 for i in n])*s
```

5.2 Q4.2

```python
from Q1 import x,y,y2
# d is the distance between samples
ct_energy = lambda x, n, d: sum([np.abs(x(i))**2 for i in n])*d

d = .0001
n = np.arange(-4,4,d)
x_en = ct_energy(x,n,d)
y1_en = ct_energy(y, n,d)
y2_en = ct_energy(y2,n,d)
```

Listing 13: Calculating energy of signals $x(t), y_1(t)$ and $y_2(t)$

```
>>> x_en: 2.6666166750070275
>>> y1_en: 1.3332833500056211
>>> y2_en: 2.6667166749880353
```

Listing 14: Output of **Listings 13**

6 Convolution

Convolution of signals $x_1[n]$ and $x_2[n]$ that symbolicly represented as $x_1[n] * x_2[n]$ is as follow

$$y[n] = \sum_{k=-\infty}^{+\infty} x_1[k]x_2[n-k] \tag{11}$$

6.1 Q5.1

The implementation of equation (11) has been Shown as one-line lambda function in **Listing 15**.

```python
# return a convolution of function x1, x2 as an function
conv = lambda x1,x2: lambda n:np.array([sum([x1(k) * x2(i-k) for k in n])for i in n])
```

Listing 15: Convolution implementation in Python

6.2 Q5.2

Consider signals $x_1[n]$ and $x_2[n]$ that has been defined below. Let's convolve them using the convolution function that we have defined in the previous section.

$$x_1[n] = (\frac{1}{2^{-n+1}}).(u[n+2] - u[n-2])$$

$$x_2[n] = \begin{cases} \sum_{k=-\infty}^{n} (sin(2k) + e^{j\pi k}).(u[k+3] - u[k-5]) & : 0 < n < 7 \\ 0 & : O.W \end{cases}$$

The implementation has been shown in **Listing 16**. After defining signals $x_1[n]$ and $x_2[n]$ they have been convolved using the **conv** function that returns the convolution as another function. and after that, they have been plotted. the result is in **Figure 13**.

```python
x1 = lambda n :(2.0)**(n-1) * unit(n)
s = lambda i: sum([(np.sin(2*k)+ np.exp(2j*np.pi*k/2))*(unit(k+3)-unit(k-5))\
                   for k in range(-3,i+1)])
x2 = lambda n : np.array([s(i) if i <7 and i>0 else 0 for i in n ])\
                 if isinstance(n,Iterable) else (s(n) if n <7 and n>0 else 0)
x3 = conv(x1,x2)

n=np.arange(-3,20,1)
gridsize = (2,2)
fig =plt.figure(figsize=(16,11))
ax1 = plt.subplot2grid(gridsize,(0,0))
ax2 = plt.subplot2grid(gridsize,(0,1))
ax3 = plt.subplot2grid(gridsize,(1,0),colspan=2)

ax1.stem(n,x1(n),use_line_collection=True)
ax1.set_title("$x_1[n]$")
ax2.stem(n,x2(n).real,use_line_collection=True)
ax2.set_title("$|x_2[n]|$")
ax3.stem(n, x3(n).real,use_line_collection=True)
ax3.set_title('|x_1[n]*x_2[n]|')

fig.show()
```

Listing 16: Convolution of $x_1(n)$ and $x_2(n)$.

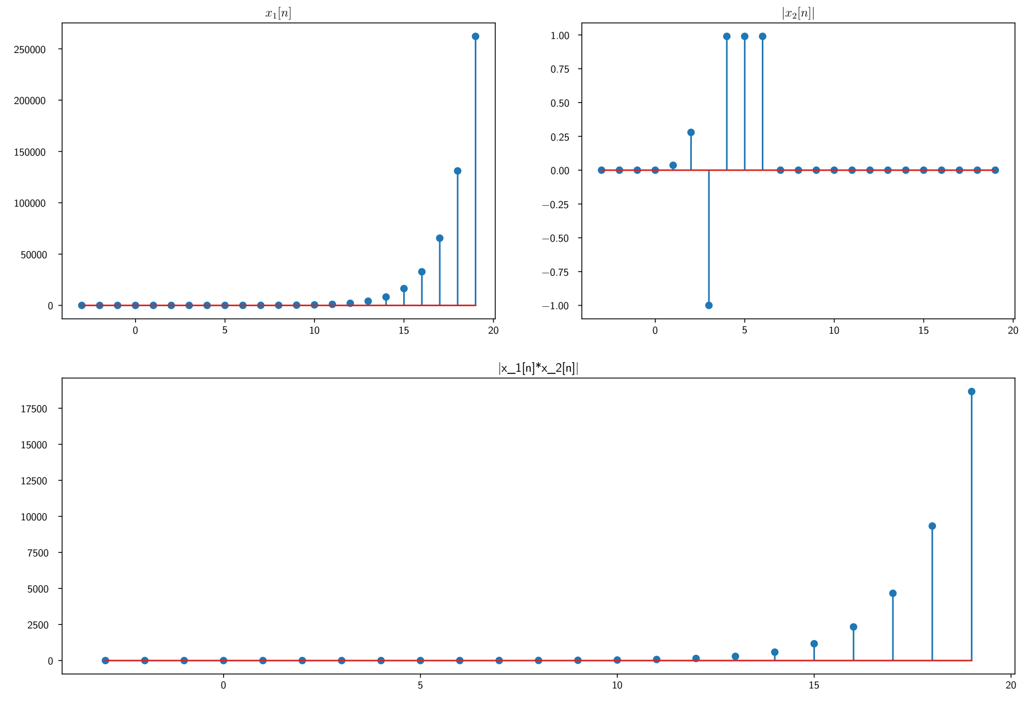

Figure 13: Result graph of the **Listing 16**

6.3 Q5.3

In this section, we implement block convolution on two signals and compare them to the simple convolution to see if it works correctly. For block convolution, we first divide the signal to blocks and then convolve them separately and sum the results. consider following $x[n]$ that we divide it to block with lenght of L.

$$x[n] = \sum_{r=0}^{\infty} x_r[n - rL] \tag{12}$$

which $P > L$ and $x_r[n]$ is as following

$$x_r[n] = \begin{cases} x[n + rL] & : 0 < n < L - 1 \\ 0 & : O.W \end{cases} \tag{13}$$

For instance, we imply this in the following signals

$$x[n] = cos(n^2)sin(\frac{2\pi n}{5})$$

$$h[n] = (0.9^n)(u[n] - u[n - 10])$$

Now first divide the signal x[n] with $L = 50$ into two blocks, first block $x_0[n]$, second block $x_1[n]$. Now we calculate $y_0[n] = x_0[n] * h[n]$ and $y_1[n] = x_1[n] * h[n]$. and using the equation below to calculate total convolution of $x[n]$ and $h[n]$.

$$y[n] = x[n] * h[n] = y_0[n] + y_1[n - L] \tag{14}$$

the implementation in Python has been shown in **Listing 17**. After defining signals they convolved using **Numpy.convolve**.

```python
# Definition of x[n]
xn = lambda n: np.cos(n**2) * np.sin(2*np.pi*n/5)
# Definition of h[n]
hn = lambda n: (.9)**n*(unit(n)- unit(n-10))
# Samples from 0 to 99
n = np.arange(0,100,1)
# Result vector of x(n) on n
x = xn(n)
# Sample for h[n]
n1 = np.arange(0,11,1)
h = hn(n1)
# Defining two block
x0n = lambda n: np.array([xn(i) if i<50 and i >=0 else 0 for i in n])
x1n = lambda n: np.array([xn(i+50) if i<50 and i >=0 else 0 for i in n])
# Result of x0 and x1 on sample n
x0 = x0n(n)
x1 = x1n(n)
yn = np.convolve(x, h)
y0 = np.convolve(x0,h)
y1 = np.convolve(x1,h)
# Sample for convolution result that has n+m-1 samples
n2 = np.concatenate((n, np.arange(n[-1],n[-1]+n1.size-1,1)))
ypn = lambda t : np.array([y0[i]+y1[i-50] if x-50>=0 else y0[i] for i,x in enumerate(t) ])
yp = ypn(n2)
```

Listing 17: Definition of signals that have been defined

In the **Listing 18** the signals that have been defined in **Listing 17**, have been plotted using **plt.stem**. The result graph has been shown in **Figure 14**. As you can see, the total convolution and the block convolution that has calculated using the equation 14 are the same.

```python
# Plotting all siganls in Grid
gridsize = (6,2)
fig = plt.figure(figsize=(16,24))
plt.subplots_adjust(hspace=0.5)
ax1 = plt.subplot2grid(gridsize,(0,0),colspan=2)
ax2 = plt.subplot2grid(gridsize,(1,0))
ax3 = plt.subplot2grid(gridsize,(2,0))
ax4 = plt.subplot2grid(gridsize, (1,1),rowspan=2)
ax5 = plt.subplot2grid(gridsize,(3,0))
ax6 = plt.subplot2grid(gridsize,(3,1))
ax7 = plt.subplot2grid(gridsize,(4,0),colspan=2)
ax8 = plt.subplot2grid(gridsize,(5,0),colspan=2)

ax1.stem(n, x)
ax1.set_title(r'$x[n]$')

ax2.stem(n, x0)
ax2.set_title(r'$x_0[n]$')
ax2.set_xlim(0,50)

ax3.stem(n, x1)
ax3.set_title(r'$x_1[n]$')
ax3.set_xlim(0,50)

ax4.stem(n1, h)
ax4.set_title(r'$h[n]$')

ax5.stem(n2,y0,'g',markerfmt='go')
ax5.set_title(r'$y_0[n]$')
ax5.set_xlim(0,60)

ax6.stem(n2,y1,'r',markerfmt='ro')
ax6.set_title(r'$y_1[n]$')
ax6.set_xlim(0,60)

ax7.stem(n2,yp,label='$y_0[n]+y_1[n]$')
ax7.stem(n2,y0,'g',markerfmt='go',label='$y_0[n]$')
ax7.stem(n2+50,y1,'r',markerfmt='ro',label='$y_1[n-50]$')
```

```python
ax7.set_xlim(0,110)
ax7.set_title(r'$y_0[n]+y_1[n]$')
ax7.legend()
ax8.stem(n2, np.convolve(x,h))
ax8.set_title(r'$x[n]*h[n]$')
```

Listing 18: Plotting signals

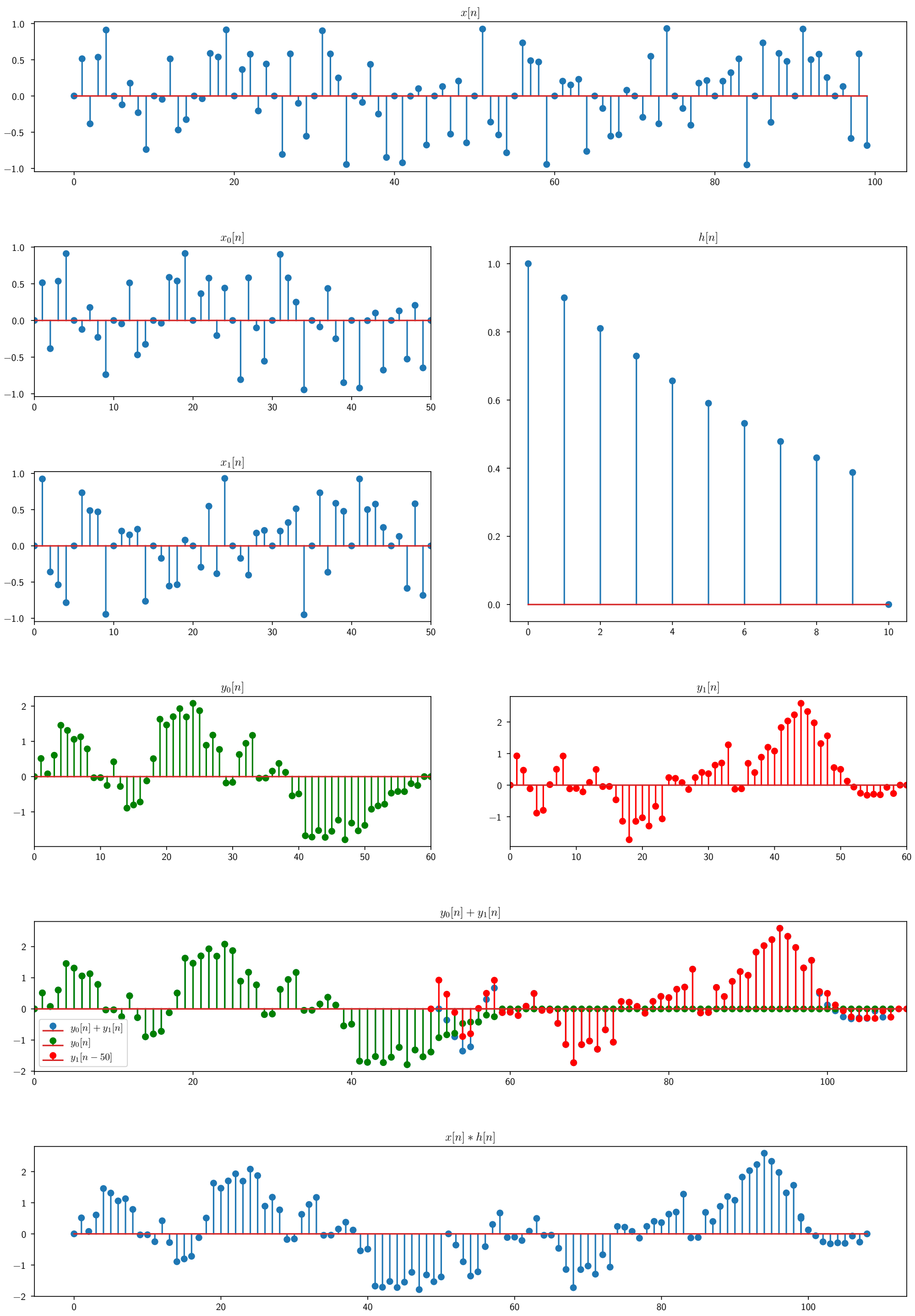

Figure 14: Result graph of the **Listing 18**