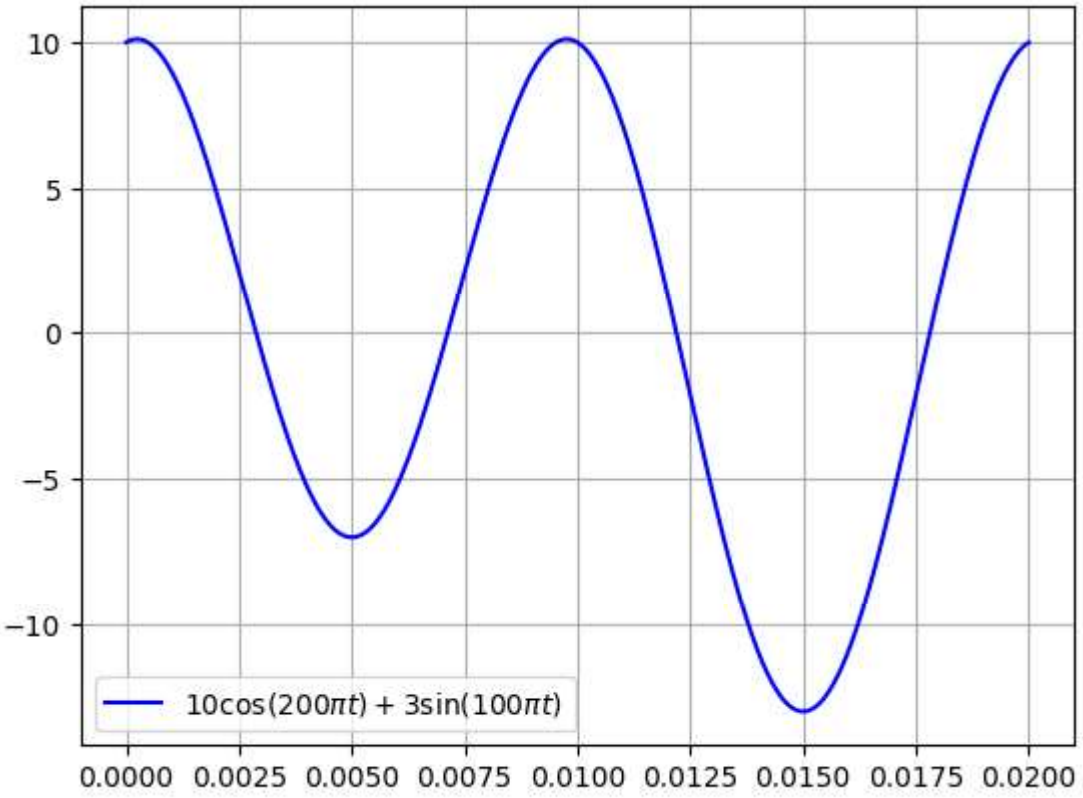
CHW1 - Kiarash Gheysari Pour - 402102302

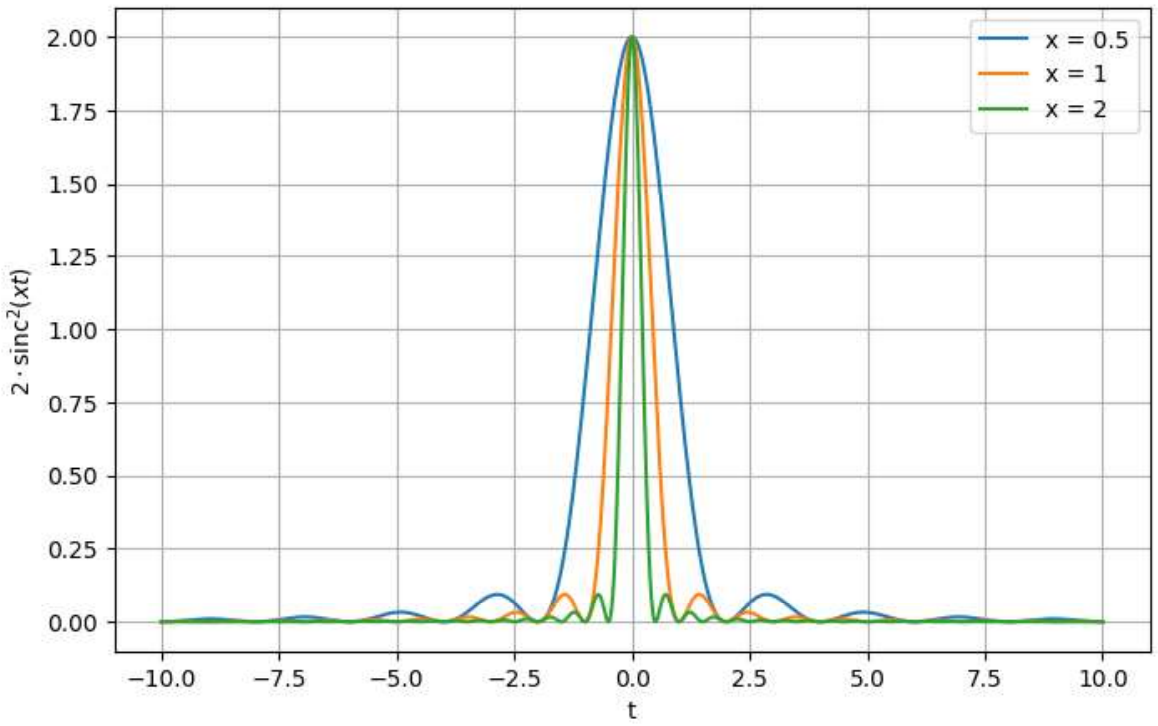
Q1.A

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt

def u(t):
    return np.where(t >= 0, 1, 0)
t = np.linspace(0, 0.02, 600)  # Time from -3 to 3 seconds
f_t = (10 * np.cos(200 * np.pi * t) + 3 * np.sin(100 * np.pi * t)) * (u(t + 2) -

plt.plot(t, f_t, label=r'$10\cos(200\pi t) + 3\sin(100\pi t)$', color='b')
plt.legend()
plt.grid()
plt.show()
```

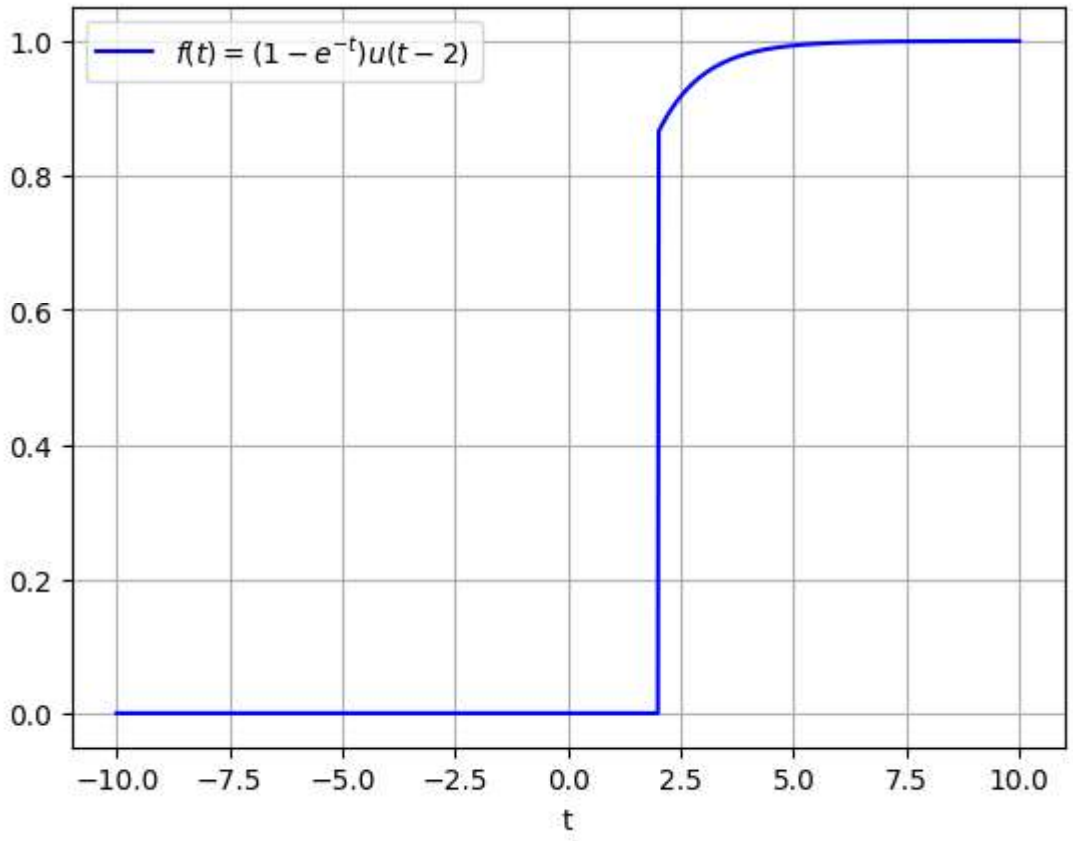


In [40]:
```
def sinc(x):
    return np.where(x == 0, 1, np.sin(np.pi * x) / (np.pi * x))

t = np.linspace(-10, 10, 1000)
x_values = [0.5, 1, 2]

plt.figure(figsize=(8, 5))

for x in x_values:
    f_t = 2 * sinc(x * t)**2
    plt.plot(t, f_t, label=f'x = {x}')

plt.xlabel('t')
plt.ylabel(r'$2 \cdot \text{sinc}^2(xt)$')
plt.legend()
```

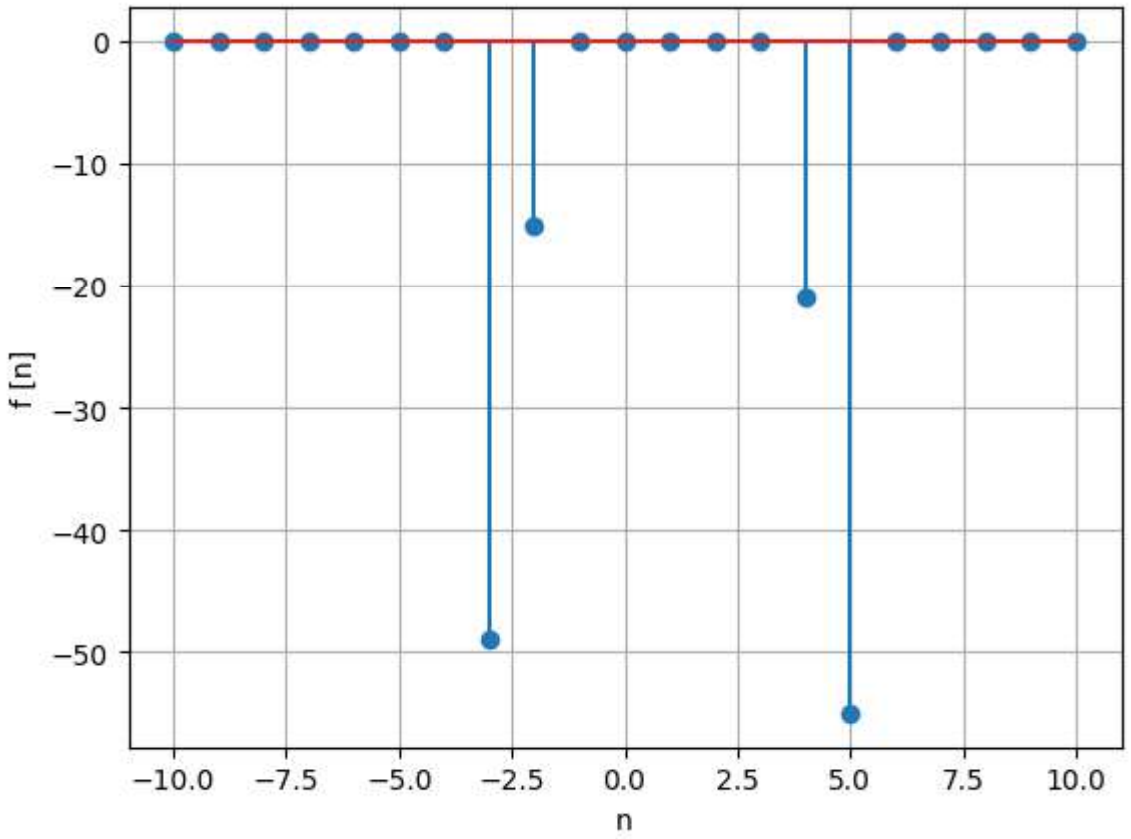
```
plt.grid()
plt.show()
```



In [4]:
```
f_t = (1 - np.exp(-t))*(u(t-2))

plt.plot(t, f_t, label=r'$f(t) = (1 - e^{-t}) u(t-2)$', color='b')
plt.xlabel('t')
plt.legend()
plt.grid()
plt.show()
```

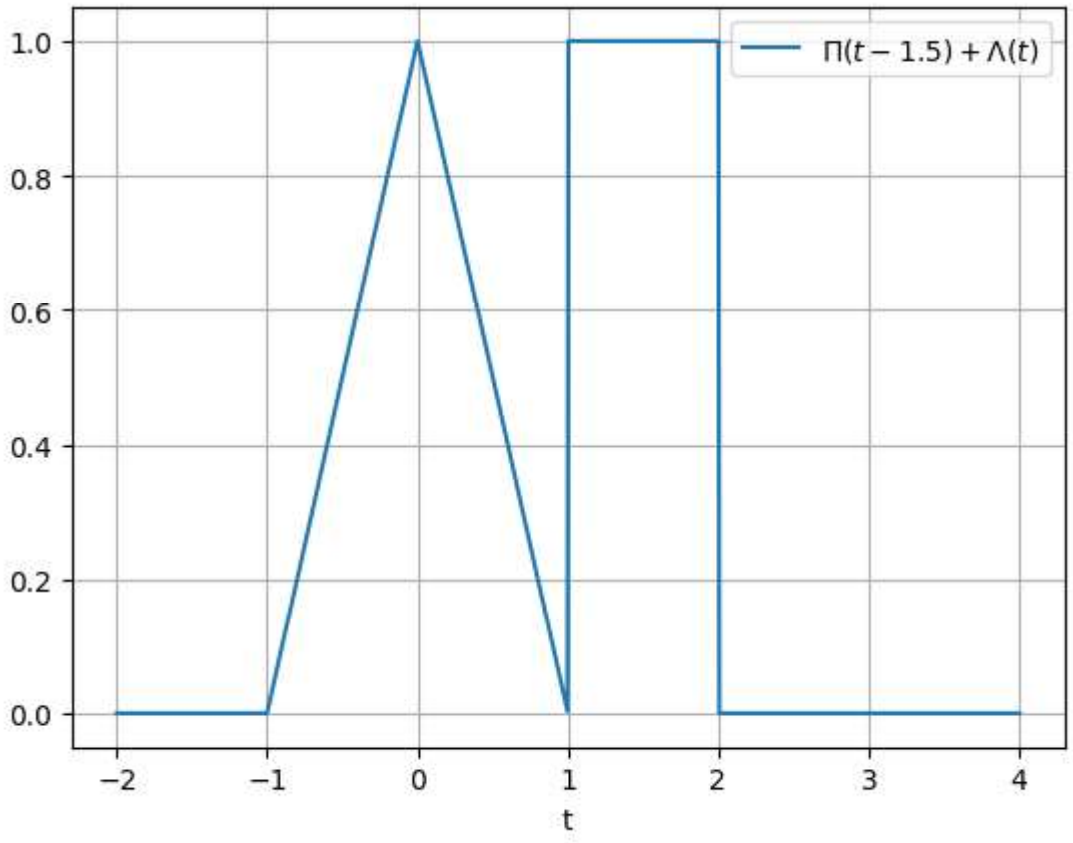


In [8]:
```python
def delta(n):
    if n==0 :
        return 1
    else :
        return 0

def f(n):
    result = 0
    for m in range(-3, 4):
        result += (n**3 - 3*n**2 + 5) * (delta(n - m) - delta(n - m - 2))
    return result

n_values = np.arange(-10, 11)
y_values = np.array([f(n) for n in n_values])

plt.stem(n_values, y_values)
plt.xlabel('n')
plt.ylabel('f [n]')
plt.grid(True)
plt.show()
```

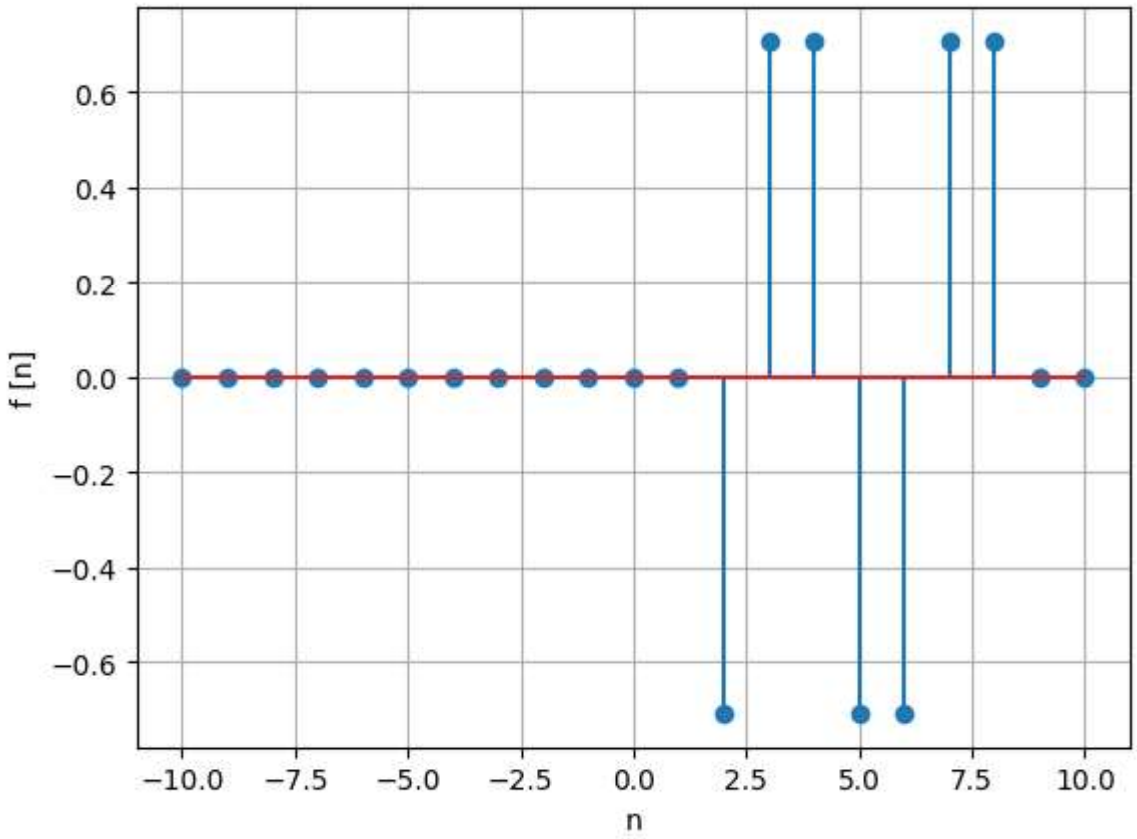


In [10]:

```
def rect(t):
    return np.where(np.abs(t) <= 0.5, 1, 0)

def triangular(t):
    return np.where(np.abs(t) <= 1, 1 - np.abs(t), 0)

def function(t):
    return rect(t - 1.5) + triangular(t)

t_values = np.linspace(-2, 4, 1000)
y_values = function(t_values)

plt.plot(t_values, y_values, label=r'$\Pi(t-1.5) + \Lambda(t)$')
plt.xlabel('t')
plt.grid(True)
plt.legend()
plt.show()
```



In [27]:
```
def f(n):
    total = 0
    for m in range(2 , 9):
        total += np.cos(np.pi*(m/2 + 0.25))*delta(n-m)
    return total

n_values = np.arange(-10, 11)
y_values = np.array([f(n) for n in n_values])

plt.stem(n_values, y_values)
plt.xlabel('n')
plt.ylabel('f [n]')
plt.grid(True)
plt.show()
```



Q1.b

We define the convolution function based on the convolution defintion :

In [15]:
```
def convolve_signals(x, h):
    len_x = len(x)
    len_h = len(h)
    len_y = len_x + len_h - 1

    y = np.zeros(len_y)
    for n in range(len_y):
        for k in range(len_x):
            if 0 <= n - k < len_h:
                y[n] += x[k] * h[n - k]
    return y
```

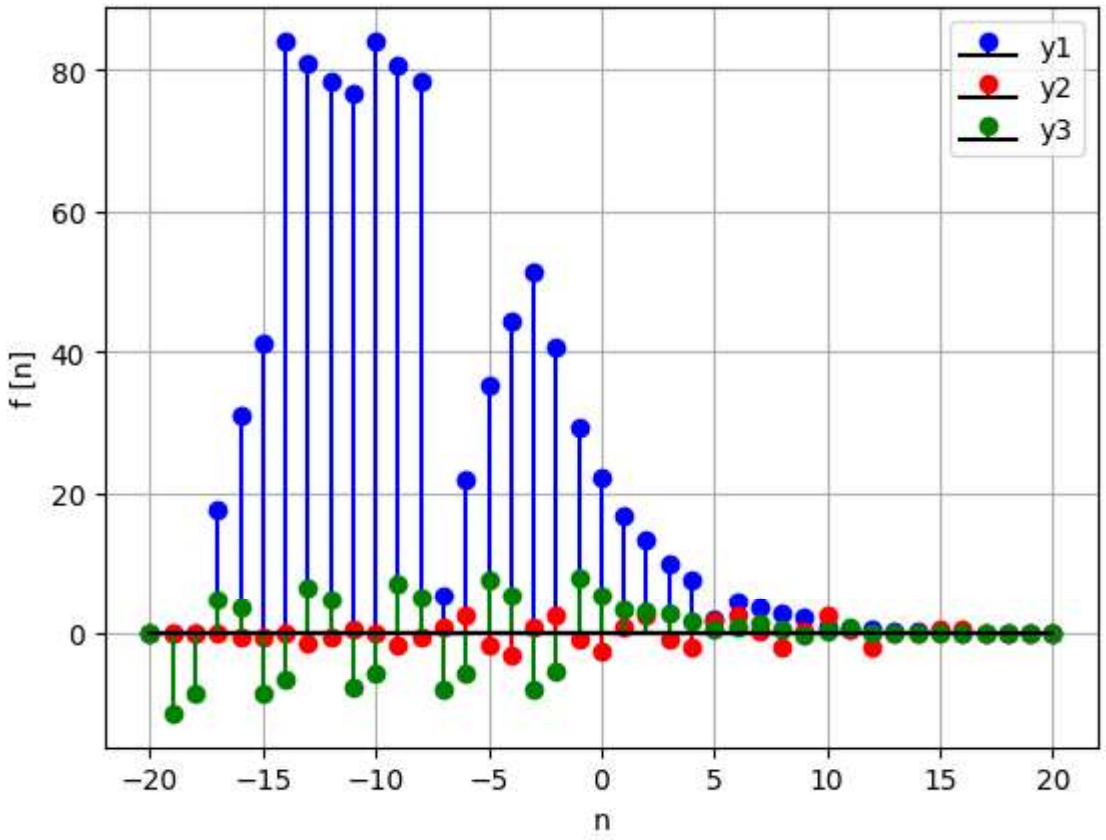
now we define our signals that we want to convolve

In [30]:
```
def x_1(n):
    return u(n+7) - u(n-8) + 0.5*delta(n) - 4*delta(n-3) +2*delta(n+4)
def x_2(n):
    return (u(n+12)-u(n-12))*(0.75)**n + delta(n-2)
def x_3(n):
    return n*sinc(n/2)*(u(n+15)-u(n-15))
```

let's convolve them !

In [41]:
```
x1 = np.array([x_1(n) for n in n_values])
x2 = np.array([x_2(n) for n in n_values])
x3 = np.array([x_3(n) for n in n_values])
```

```
y1 = convolve_signals(x1 , x2)
y2 = convolve_signals(x1 , x3)
y3 = convolve_signals(x2 , x3)

n2_values = range(-20,21)

plt.stem(n2_values, y1, linefmt='b-', markerfmt='bo', basefmt='k')
plt.stem(n2_values, y2, linefmt='r-', markerfmt='ro', basefmt='k')
plt.stem(n2_values, y3, linefmt='g-', markerfmt='go', basefmt='k')

plt.xlabel('n')
plt.ylabel('f [n]')
plt.grid(True)
plt.legend(['y1', 'y2', 'y3'])
plt.show()
```

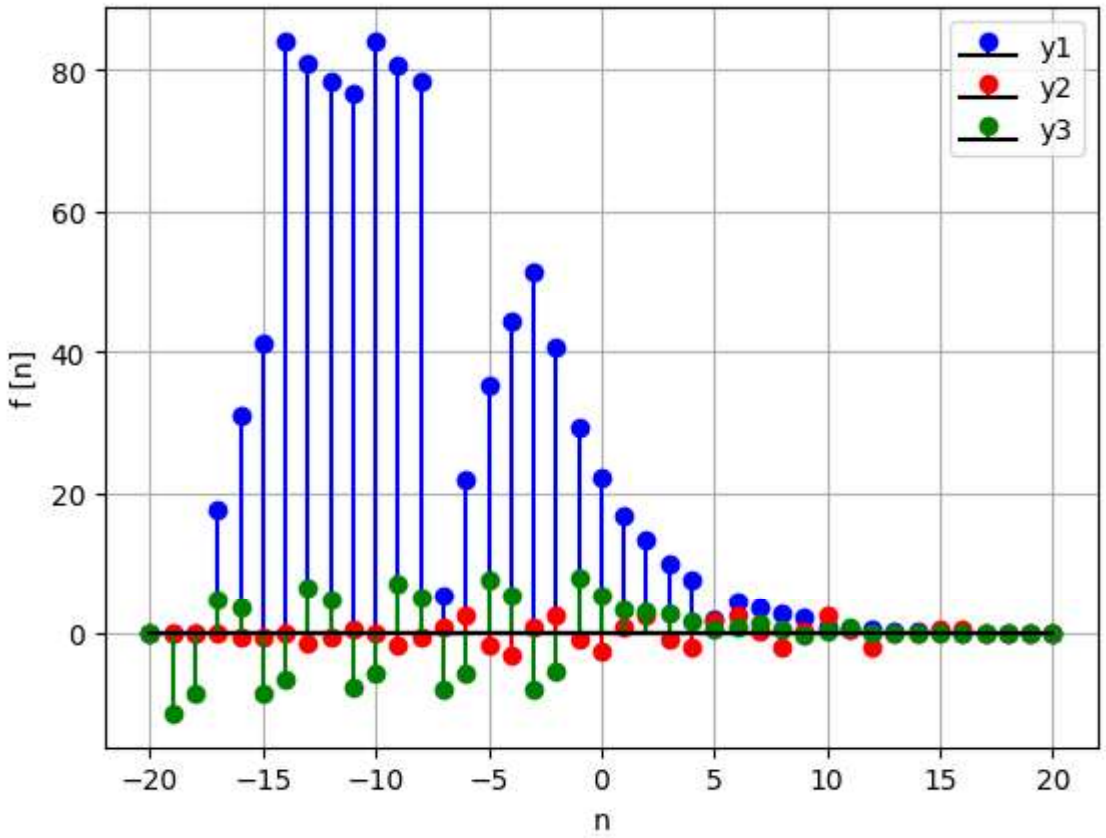


let's convolve them using the np.convolve() method too , just to make sure our function works fine!

In [ ]:

```
y1 = np.convolve(x1 , x2)
y2 = np.convolve(x1 , x3)
y3 = np.convolve(x2 , x3)

plt.stem(n2_values, y1, linefmt='b-', markerfmt='bo', basefmt='k')
plt.stem(n2_values, y2, linefmt='r-', markerfmt='ro', basefmt='k')
plt.stem(n2_values, y3, linefmt='g-', markerfmt='go', basefmt='k')

plt.xlabel('n')
plt.ylabel('f [n]')
plt.grid(True)
```

```
plt.legend(['y1', 'y2', 'y3'])
plt.show()
```



Seems like we've done a good job :D

Now let's take this "convolution" game to 2D !!

In [44]:
```
def convolve2d(image, kernel):
    i_h, i_w = image.shape
    k_h, k_w = kernel.shape

    pad_h = (k_h - 1) // 2
    pad_w = (k_w - 1) // 2

    padded_image = np.pad(image, ((pad_h, pad_h), (pad_w, pad_w)), mode='constan
    output = np.zeros((i_h, i_w))

    kernel_flipped = np.flipud(np.fliplr(kernel))

    for i in range(i_h):
        for j in range(i_w):
            region = padded_image[i:i+k_h, j:j+k_w]
            output[i, j] = np.sum(region * kernel_flipped)

    return output
```

because we want to use our function later in image proccesing , I've implemented a zero padded version of 2d convolution to maintain the image size.

now let's convolve some signals!

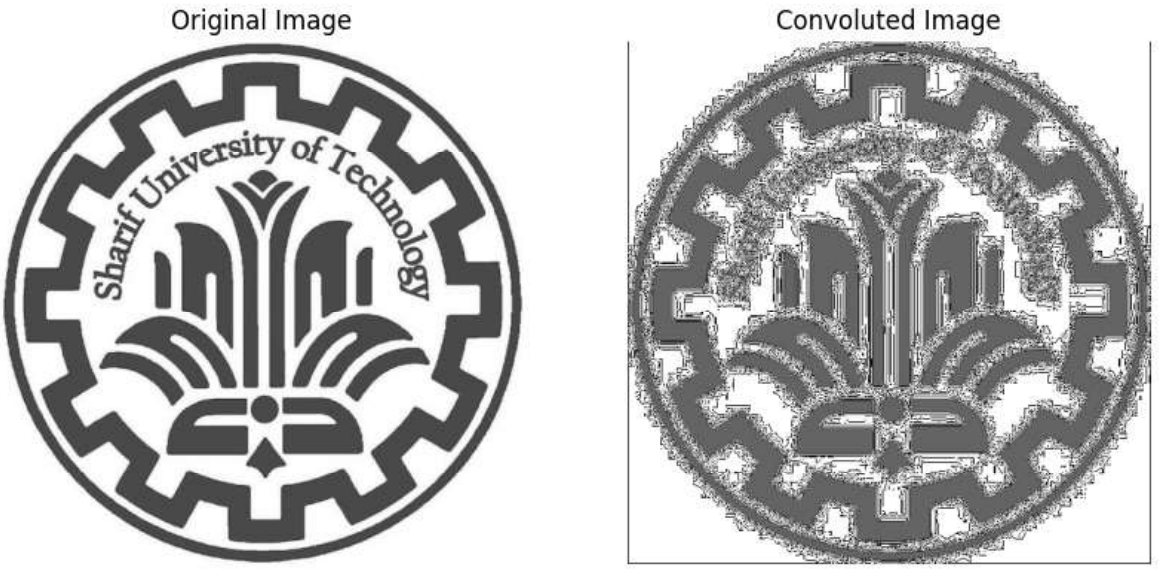
In [50]:
```
h1 = np.array([[1,0,1],
               [0,1,0],
               [0,0,1]])
h2 = np.array([[-1,-1,-1],
               [-1,8,-1],
               [-1,-1,-1]])
x = np.array([
    [25, 100, 75, 49, 130],
    [50, 80, 0, 70, 100],
    [5, 10, 20, 30, 0],
    [60, 50, 12, 24, 32],
    [37, 53, 55, 21, 90],
    [140, 17, 0, 23, 222]
])

y1 = convolve2d(x , h1)
y2 = convolve2d(x , h2)

print('first signal :' )
print(y1)
print('\n')
print('second signal :' )
print(y2)
```

```
first signal :
[[105. 150. 225. 149. 200.]
 [ 60. 130. 140. 165. 179.]
 [ 55. 132. 174.  74.  94.]
 [113. 147.  96. 189.  83.]
 [ 54. 253. 145. 255. 137.]
 [140.  54.  53.  78. 243.]]


second signal :
[[ -30.  570.  301.   17.  821.]
 [ 180.  355. -434.  156.  521.]
 [-210. -197. -116.  -18. -256.]
 [ 325.  148. -167.  -68.   91.]
 [ -24.   53.  240. -290.  398.]
 [1013. -149. -169. -204. 1642.]]
```

let's apply some edge detection to our beloved universitys logo!

In [58]:
```
from PIL import Image

image = Image.open('sharif_logo.jpg').convert('L')
image_np = np.array(image)

kernel = np.array([[1, 1, 1],
                   [1, -7, 1],
                   [1, 1, 1]])

convoluted_image = convolve2d(image_np, kernel)

output_image = Image.fromarray(convoluted_image.astype(np.uint8))

plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)
```

```
plt.imshow(image, cmap='gray')
plt.title('Original Image')
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(output_image, cmap='gray')
plt.title('Convoluted Image')
plt.axis('off')

plt.show()
```

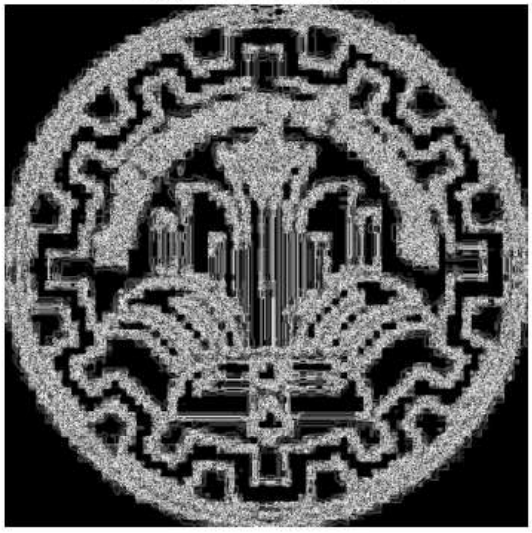


now lets apply the built in edge detection method that scipy provides us with and see how it hold against our manual convolving method!

In [60]:
```
from scipy.ndimage import sobel

edge_x = sobel(image_np, axis=0)
edge_y = sobel(image_np, axis=1)

edge_magnitude = np.sqrt(edge_x**2 + edge_y**2)

edge_magnitude = np.uint8(np.clip(edge_magnitude, 0, 255))

output_image = Image.fromarray(edge_magnitude)

plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)
plt.imshow(image, cmap='gray')
plt.title('Original Image')
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(output_image, cmap='gray')
plt.title('Edge Detected Image')
plt.axis('off')

plt.show()
```

Original Image

Edge Detected Image



I'm no expert in image proccessing but it seems like that the scipy library has beaten us !

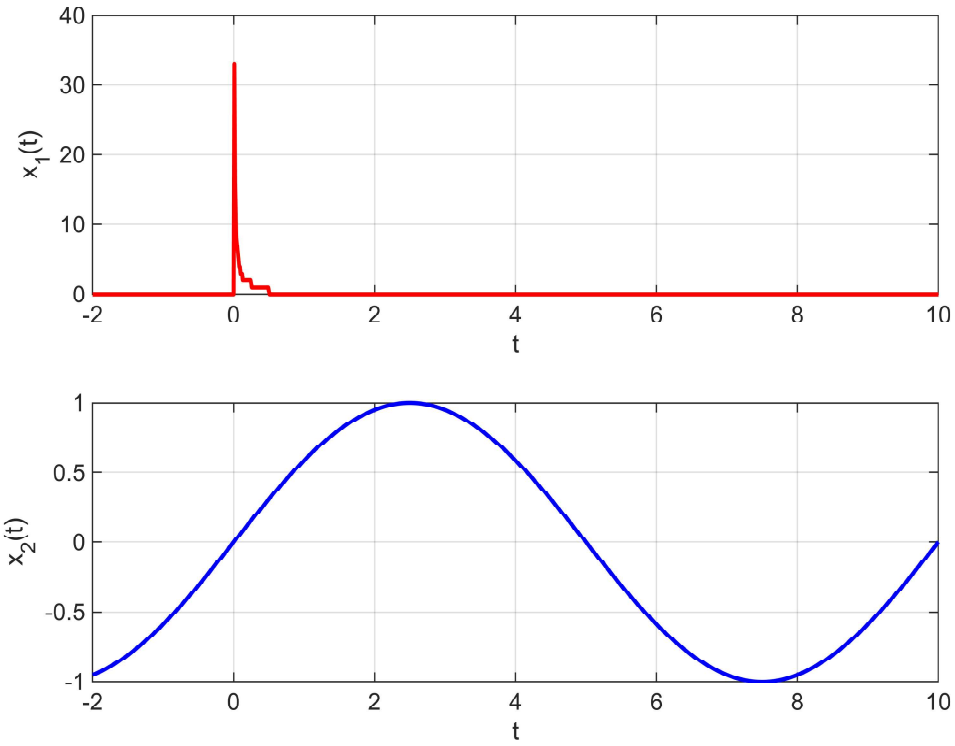
no worries , we'll get them next time inshallah

Q2)

we've defined our systems in another class called Systems and we'll use them throughout this code.

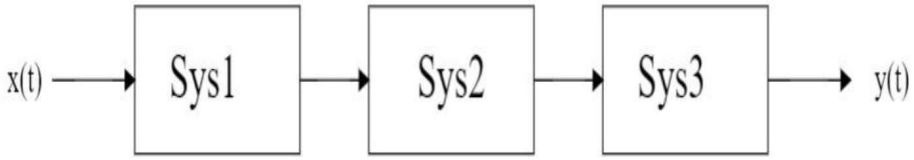
now we create our signals and see how they look like.

```
t = -2:0.01:10;
x1 = zeros(size(t));
for n = 0:100
    x1 = x1 + (heaviside(6*n*t - 1) - heaviside(6*n*t - 3));
end

x2 = sin(0.2 * pi * t);

figure;

subplot(2,1,1);
plot(t, x1, 'r', 'LineWidth', 1.5);
xlabel('t'); ylabel('x_1(t)');
grid on;

subplot(2,1,2);
plot(t, x2, 'b', 'LineWidth', 1.5);
xlabel('t'); ylabel('x_2(t)');
grid on;
```

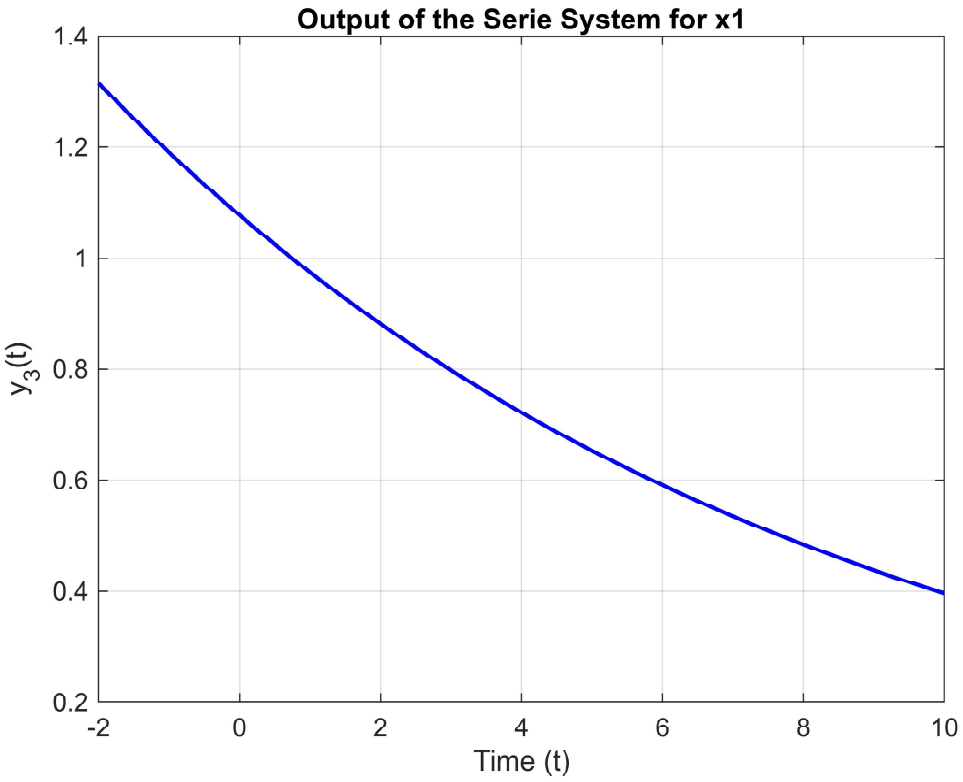


Now we feed our signals to our Systems to see what they look like!

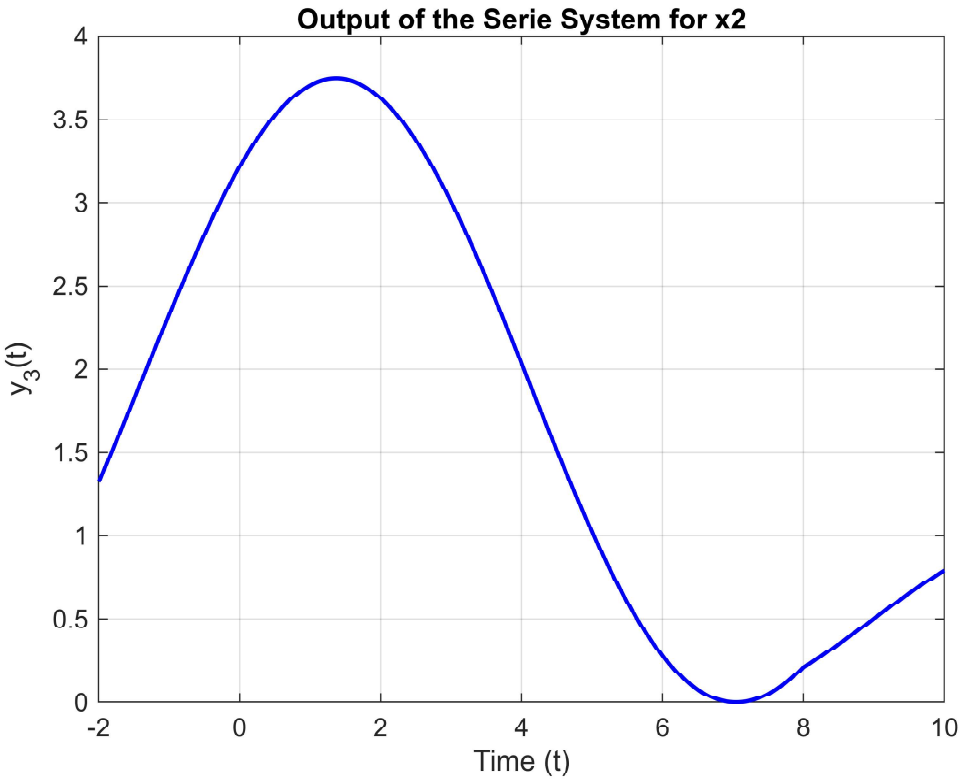
the first System configuration that we'll simulate is the "Serie" configuration :



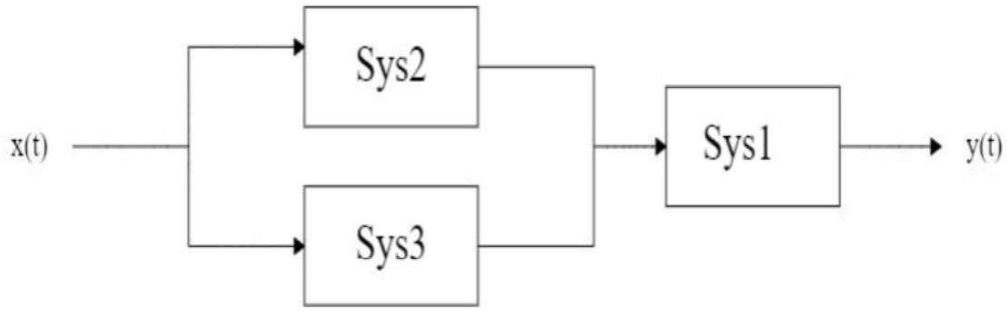

```
% first System (Series Configuration)
% for signal x1
y1 = Systems.system1(t , x1);
y2 = Systems.system2(y1);
y3 = Systems.system3(t , y2);

figure;
plot(t, y3, 'b', 'LineWidth', 1.5);
xlabel('Time (t)');
ylabel('y_3(t)');
title('Output of the Serie System for x1');
grid on;
```




```
% first System (Series Configuration)
% for signal x2
y1 = Systems.system1(t , x2);
y2 = Systems.system2(y1);
y3 = Systems.system3(t , y2);

figure;
plot(t, y3, 'b', 'LineWidth', 1.5);
xlabel('Time (t)');
ylabel('y_3(t)');
title('Output of the Serie System for x2');
grid on;
```

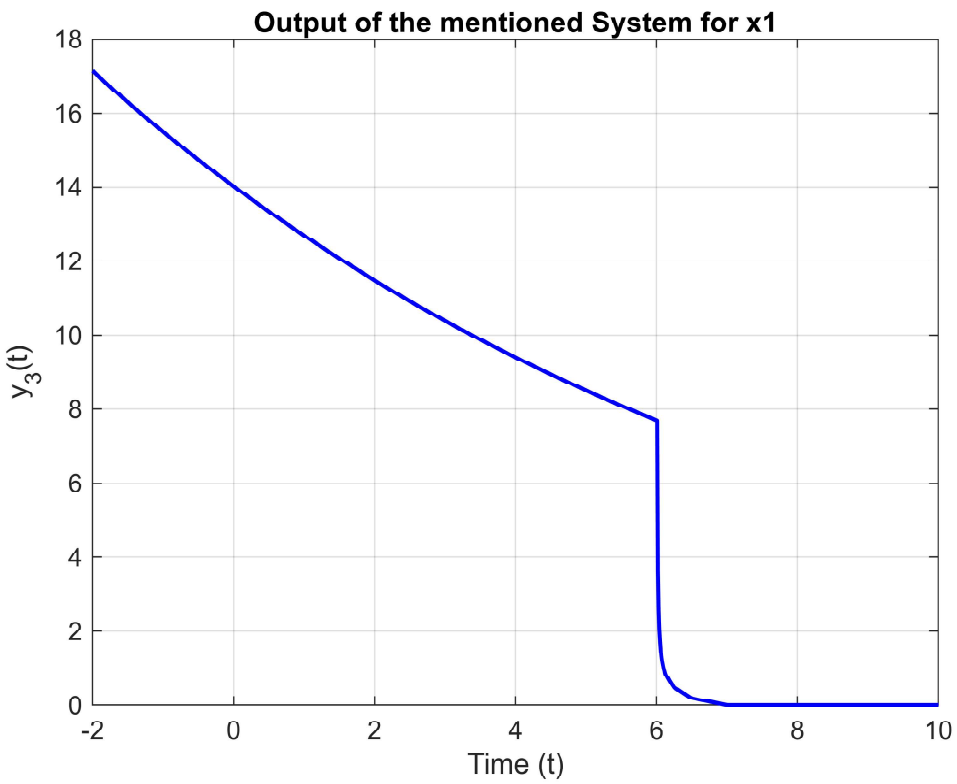


Now we implement this configuration :

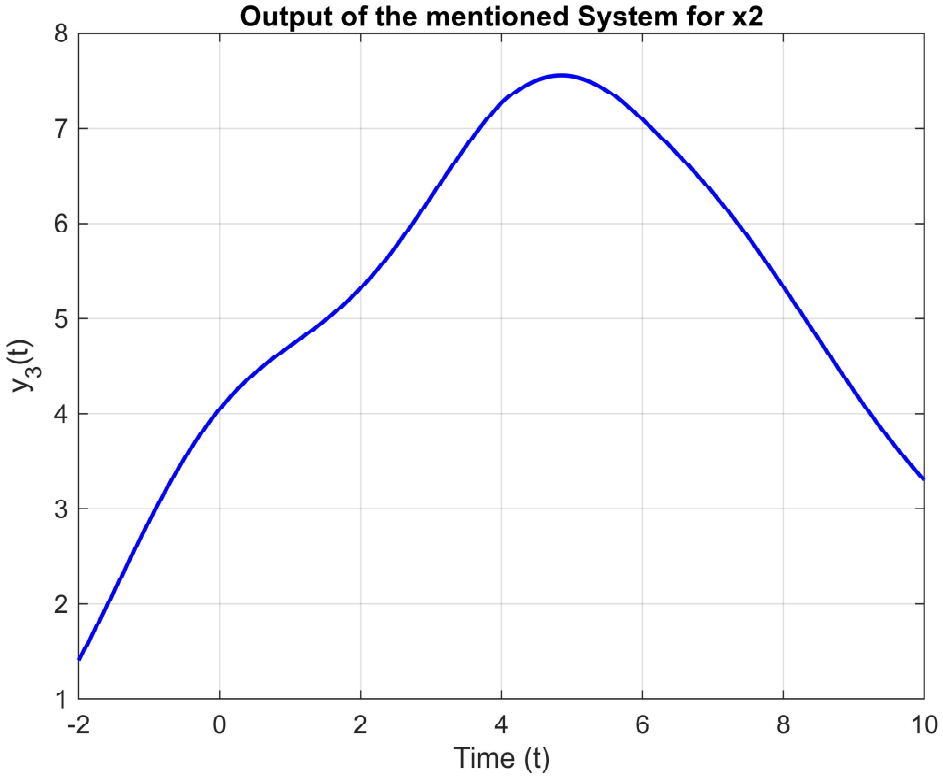



```
y1 = Systems.system3(t , x1);
y2 = Systems.system2(x1);
y3 = Systems.system1(t , y1 + y2);

figure;
plot(t, y3, 'b', 'LineWidth', 1.5);
xlabel('Time (t)');
ylabel('y_3(t)');
```

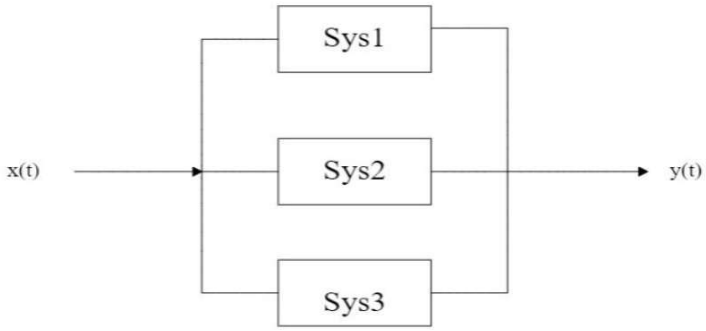
```
title('Output of the mentioned System for x1');
grid on;
```




```
y1 = Systems.system3(t , x2);
y2 = Systems.system2(x2);
y3 = Systems.system1(t , y1 + y2);

figure;
plot(t, y3, 'b', 'LineWidth', 1.5);
xlabel('Time (t)');
ylabel('y_3(t)');
title('Output of the mentioned System for x2');
grid on;
```

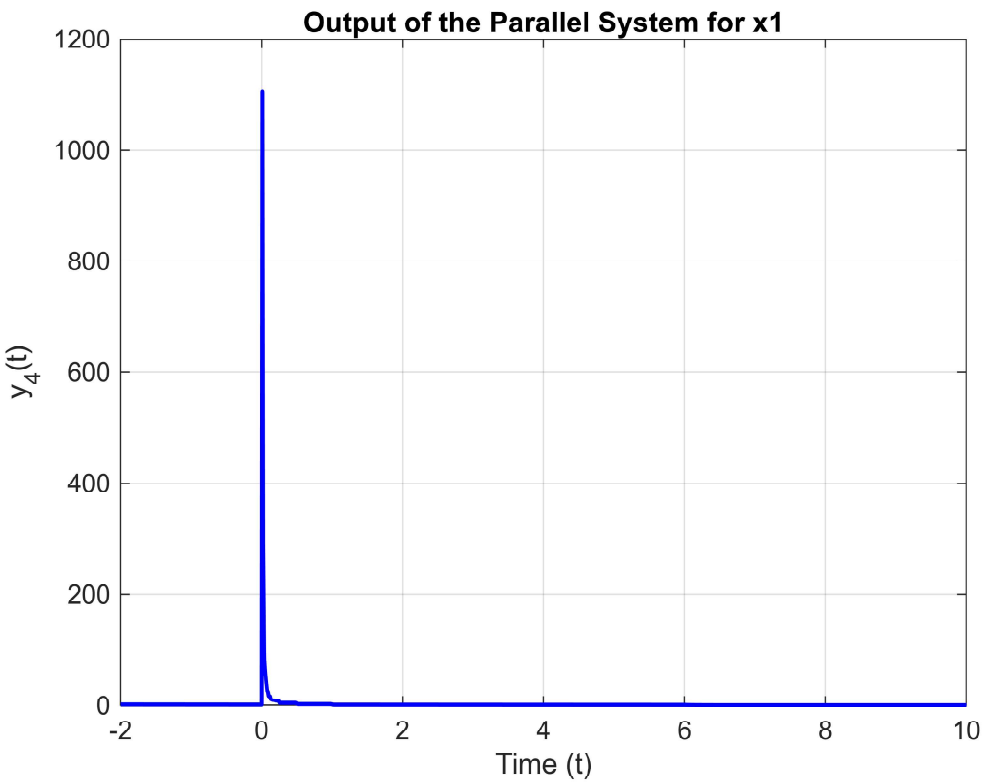


Now we implement this System :

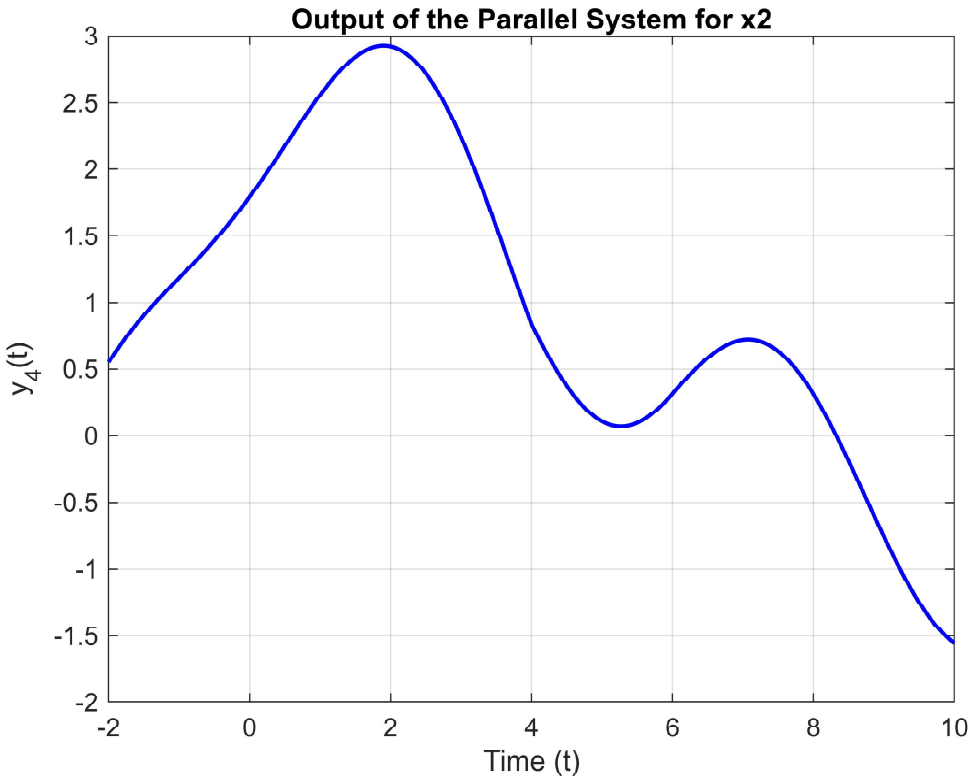



```
y1 = Systems.system3(t , x1);
y2 = Systems.system2(x1);
y3 = Systems.system1(t , x1);
y4 = y1 + y2 + y3;

figure;
plot(t, y4, 'b', 'LineWidth', 1.5);
xlabel('Time (t)');
ylabel('y_4(t)');
```

```
title('Output of the Parallel System for x1');
grid on;
```




```
y1 = Systems.system3(t , x2);
y2 = Systems.system2(x2);
y3 = Systems.system1(t , x2);
y4 = y1 + y2 + y3;

figure;
plot(t, y4, 'b', 'LineWidth', 1.5);
xlabel('Time (t)');
ylabel('y_4(t)');
title('Output of the Parallel System for x2');
grid on;
```


Output of the Parallel System for x2
3
2.5
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
y4(t)
-2
0
2
4
6
8
10
Time (t)

Q3 :

the code below implements the logic for our random walk (the code used for creating the gif is in the zip in a file named 3.p)

In [ ]:

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

def random_motion(n, steps=50):
    grid = np.random.randint(1, 11, (n, n))
    x, y = np.random.randint(0, n, size=2)

    prev_x, prev_y = None, None

    fig, ax = plt.subplots()

    def update(frame):
        nonlocal x, y, prev_x, prev_y
        ax.clear()
        ax.imshow(grid, cmap='gray', alpha=0.5)
        ax.scatter(y, x, c='red', s=100)
        ax.set_xticks([])
        ax.set_yticks([])
        ax.set_title(f"Step {frame + 1}")

        neighbors = []
        weights = []

        for dx, dy in [(-1,0), (1,0), (0,-1), (0,1), (-1,-1), (-1,1), (1,-1), (1
            nx, ny = x + dx, y + dy
            if 0 <= nx < n and 0 <= ny < n and (nx, ny) != (prev_x, prev_y):
                neighbors.append((nx, ny))
                weights.append(grid[nx, ny])

        if neighbors:
            next_x, next_y = random.choices(neighbors, weights=weights)[0]
            prev_x, prev_y = x, y
            x, y = next_x, next_y

    ani = animation.FuncAnimation(fig, update, frames=steps, interval=300)
    plt.show()

random_motion(n=8, steps=50)
```