UNIVERSIDADE
FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

QXD0144 - SINAIS E SISTEMAS

Decomposição e Convolução de Sinais de Tempo Discreto em *Python*

Aluno: Pedro Henrique Magalhães Botelho

Matrícula: 471047

Orientador: André Ribeiro Braga

GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO - 6º SEMESTRE

QUIXADÁ - CE
2022

Conteúdo

1 Introdução

Esse é o trabalho prático da disciplina de Sinais e Sistemas, com supervisão professor André R. Braga, que consiste na implementação de uma ferramenta computacional que realize a decomposição de sinais de tempo discreto em sinais pares e ímpares, bem como a soma da convolução de um sinal de entrada com uma resposta a um impulso.

A ferramenta consiste de dois códigos, com as funções implementadas, bem como dos exemplos, que utilizam das funções implementadas para traçar os gráficos e salvar as imagens. A ferramenta pode ser acessada pelo repositório no GitHub. Lá temos alguns arquivos. Na pasta "convolution_sum/" o arquivo "convolution.py" traz as funções para possibilitar a soma da convolução entre dois sinais, e a pasta "signal_decomposition/" o arquivo "decomposition.py", com funções para possibilitar "dividir" o sinal em dois sinais diferentes.

Algumas funções existem nos dois arquivos, como: unitImpulse(n), unitStep(n), plotSignal(domain, signal, title) e plotSignalAndSave(domain, signal, title, fileName). Falaremos delas em breve. O código traz bastante explicações sobre as implementações e os sinais.

2 Decomposição de Sinais

Foram utilizadas as fórmulas padrões para a decomposição do sinal em duas partes. O livro "Sinais e Sistemas" do Alan Oppenheim traz essas equações na página 10. Tendo em mente que $x[n]$ é o sinal de entrada do sistema, temos que:

$$\mathcal{E}v\{x[n]\} = \frac{1}{2}(x[n] + x[-n]) \tag{1}$$

$$\mathcal{O}d\{x[n]\} = \frac{1}{2}(x[n] - x[-n]) \tag{2}$$

Onde $\mathcal{E}v\{x[n]\}$ diz respeito à parte par do sinal e $\mathcal{O}d\{x[n]\}$ à parte ímpar. Podemos decompor o sinal dividindo ele em duas partes, assim como podemos recompor o sinal somando ambas as partes. Dessa forma, voltamos a ter o sinal completo novamente.

2.1 Implementação da Ferramenta

A implantação da ferramenta foi utilizando a linguagem *Python*, assim como bilbiotecas para facilitar o tracejamento dos gráficos. São elas: *NumPy*, *Matplotlib* e *PyLab*. Vamos discutir cada função:

A função abaixo retorna um valor representando um impulso unitário. Dependendo dos valores em n o valor retornado muda, sendo n uma lista de valores representando o domínio de

um sinal. Se $n = 0$, temos resultado 1, e caso contrário, 0.

```
def unitImpulse(n):
    return np.where(n == 0, 1, 0)
```

Já a função abaixo retorna um degrau unitário, dependendo dos valores em n. Muito parecida com a função anterior, esta retorna 1 quando n é maior ou igual a 0, e retorna 0 caso contrário. O método **where(condition, [x, y])** nos ajuda aqui, já que podemos retornar alguns valores, dependendo se os valores de uma lista cumprem uma condição ou não.

```
def unitStep(n):
    return np.where(n >= 0, 1, 0)
```

As duas funções abaixo realizam serviços muito semelhantes: traçar um gráfico na tela, dado um sinal, um domínio e um título para o gráfico. A função abaixo apenas traça o gráfico e nada mais.

```
def plotSignal(domain, signal, title):
    fig = figure()
    fig.set_size_inches(16, 10)

    signalPlot = fig.add_subplot()
    signalPlot.set_title(title)
    signalPlot.set_xlabel("$n$ (Domain)")
    signalPlot.stem(domain, signal)
    show()
```

Já a função abaixo além de traçar o gráfico também salva o arquivo no caminho informado.

```
def plotSignalAndSave(domain, signal, title, fileName):
    fig = figure()
    fig.set_size_inches(16, 10)

    signalPlot = fig.add_subplot()
    signalPlot.set_title(title)
    signalPlot.set_xlabel("$n$ (Domain)")
    signalPlot.stem(domain, signal)
    fig.savefig(fileName, dpi=fig.dpi)
    show()
```

Todas essa funções estavam nos dois arquivos. Temos mais duas funções para ver, dessa vez próprias da ferramenta de decomposição:

A função abaixo retorna a parte par do sinal informado. Perceba a similaridade com a fórmula dada no começo da seção 2.

```
def decomposeEven(signal, n):
    return 0.5*(signal(n) + signal(-n))
```

A função abaixo retorna a parte ímpar do sinal informado. Novamente, A implementação computacional representa bem a fórmula geral, dada no começo da seção 2.

```
def decomposeOdd(signal, n):
    return 0.5*(signal(n) - signal(-n))
```

Com essas funções podemos realizar muitas operações com sinais, como decompor e recompor sinais, podendo os mostrar na tela ao final do programa. Vamos agora ver alguns exemplos utilizando a ferramenta.

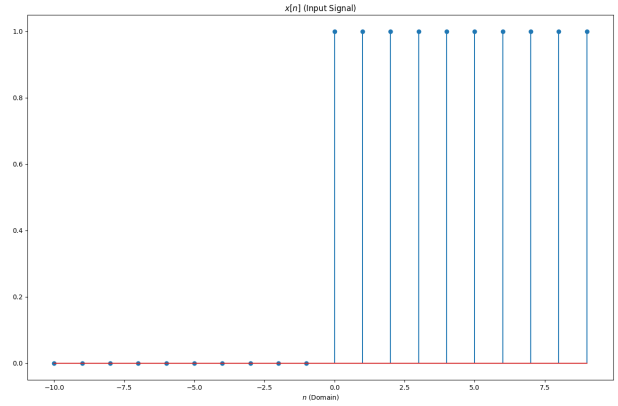
2.2 Exemplos de Sinais

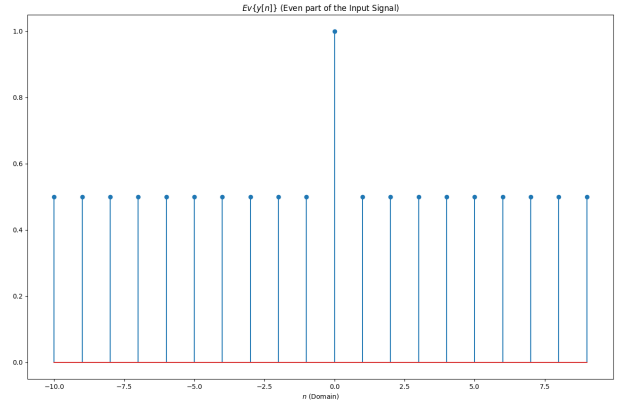
Foram utilizados três sinais para exemplificar as funções da ferramenta. Os sinais foram retirados do livro "Sinais e Sistemas" do Alan Oppenheim, segunda edição. Segue os códigos e as imagens respectivas.

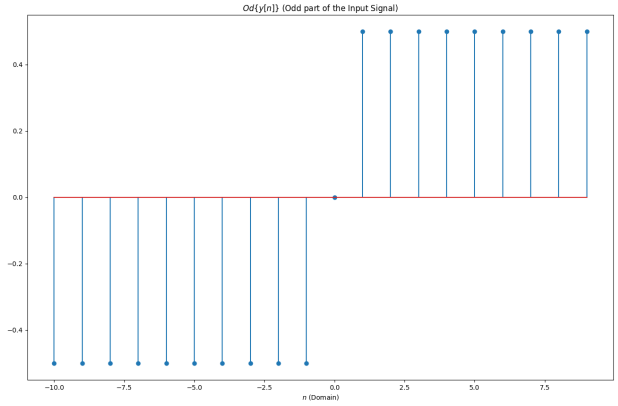
2.2.1 Sinal da Figura 1.18

A figura 1.18 traz um sinal simples, mas que permite que nós tracemos de forma satisfatória um gráfico para testarmos a ferramenta. Perceba que, após incluirmos as bibliotecas externas, definimos um sinal de entrada, que é um degrau unitário, e logo decompomos o sinal usando funções definidas anteriormente. O sinal é então recomposto, pela soma do par com o ímpar, e os quatros sinais(original, par, ímpar e soma) são mostrados na tela e salvos na pasta imagens.

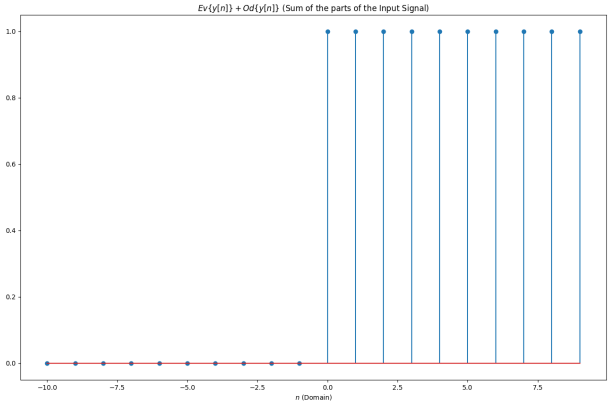
```
import numpy as np
from decomposition import *

def inputSignal(n):
    return unitStep(n)

n = np.arange(-10, 10)

evenSignal = decomposeEven(inputSignal, n)
oddSignal = decomposeOdd(inputSignal, n)
signalSum = evenSignal + oddSignal

plotSignalAndSave(n, inputSignal(n), "$x[n]$ (Input Signal)", "images/
    first_signal_input.png")
plotSignalAndSave(n, evenSignal, "$Ev\{y[n]\}$ (Even part of the Input Signal)
    ", "images/first_signal_even.png")
plotSignalAndSave(n, oddSignal, "$Od\{y[n]\}$ (Odd part of the Input Signal)",
     "images/first_signal_odd.png")
plotSignalAndSave(n, signalSum, "$Ev\{y[n]\} + Od\{y[n]\}$ (Sum of the parts
    of the Input Signal)", "images/first_signal_sum.png")
```







Perceba, na figura abaixo, que a soma dos sinais é igual ao sinal original!

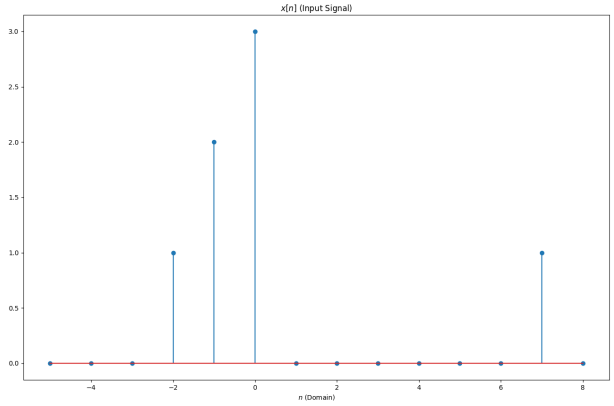


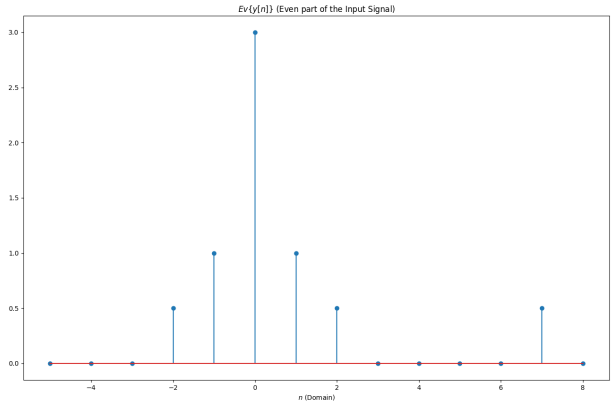
2.2.2 Sinal da Questão 1.24b

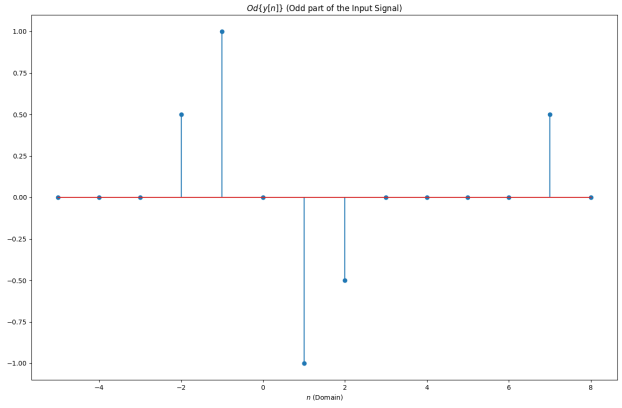
Esse sinal também é bem simples, porém também é bem interessante para utilizarmos a ferramenta.

```
import numpy as np
from decomposition import *

def inputSignal(n):
    return 1*unitImpulse(n + 2) + 2*unitImpulse(n + 1) + 3*unitImpulse(n) + 1*
        unitImpulse(n - 7)

n = np.arange(-5, 9)

evenSignal = decomposeEven(inputSignal, n)
oddSignal = decomposeOdd(inputSignal, n)
signalSum = evenSignal + oddSignal

plotSignalAndSave(n, inputSignal(n), "$x[n]$ (Input Signal)", "images/
    second_signal_input.png")
plotSignalAndSave(n, evenSignal, "$Ev\{y[n]\}$ (Even part of the Input Signal)
    ", "images/second_signal_even.png")
plotSignalAndSave(n, oddSignal, "$Od\{y[n]\}$ (Odd part of the Input Signal)",
     "images/second_signal_odd.png")
plotSignalAndSave(n, signalSum, "$Ev\{y[n]\} + Od\{y[n]\}$ (Sum of the parts
    of the Input Signal)", "images/second_signal_sum.png")
```

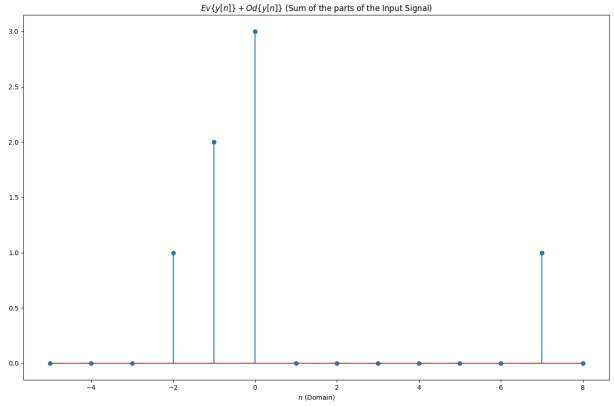
Temos abaixo as imagens relativas ao código:





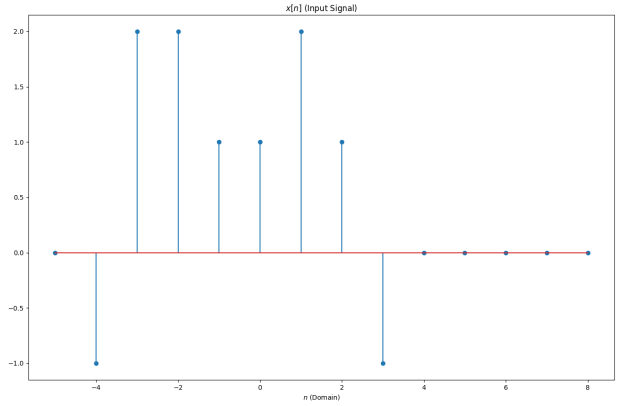


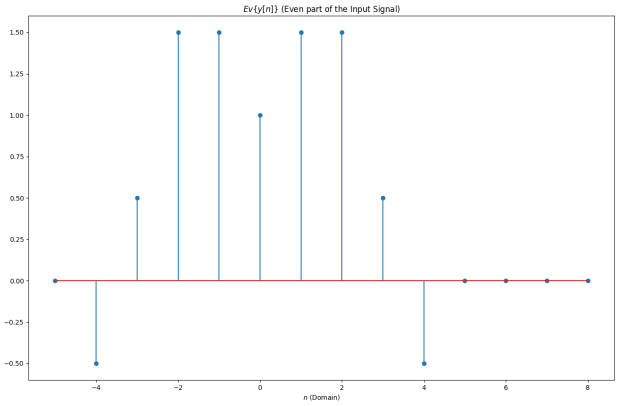
Temos abaixo, então, a soma dos sinais:

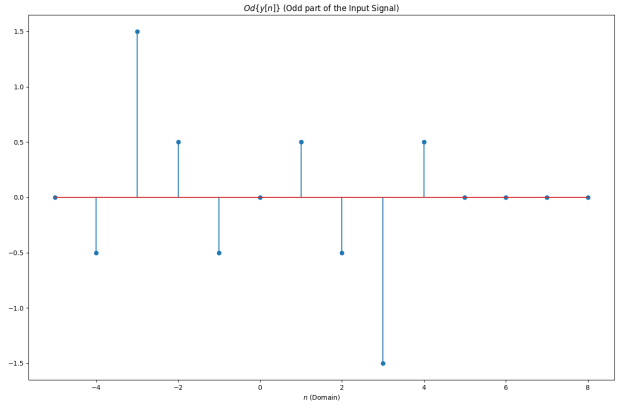


2.2.3 Sinal da Questão 1.24c

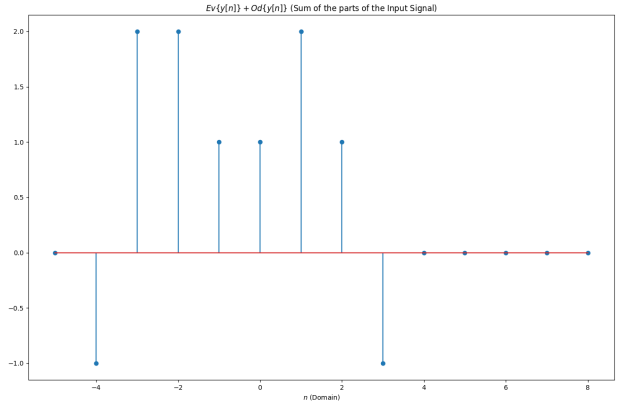
```
import numpy as np
from decomposition import *

def inputSignal(n):
    return (-1)*unitImpulse(n + 4) + 2*unitImpulse(n + 3) + 2*unitImpulse(n +
        2) + 1*unitImpulse(n + 1)  + 1*unitImpulse(n) + 2*unitImpulse(n - 1) +
        1*unitImpulse(n - 2) -  1*unitImpulse(n - 3)

n = np.arange(-5, 9)

evenSignal = decomposeEven(inputSignal, n)
oddSignal = decomposeOdd(inputSignal, n)
signalSum = evenSignal + oddSignal

plotSignalAndSave(n, inputSignal(n), "$x[n]$ (Input Signal)", "images/
    third_signal_input.png")
plotSignalAndSave(n, evenSignal, "$Ev\{y[n]\}$ (Even part of the Input Signal)
    ", "images/third_signal_even.png")
plotSignalAndSave(n, oddSignal, "$Od\{y[n]\}$ (Odd part of the Input Signal)",
     "images/third_signal_odd.png")
plotSignalAndSave(n, signalSum, "$Ev\{y[n]\} + Od\{y[n]\}$ (Sum of the parts
    of the Input Signal)", "images/third_signal_sum.png")
```







Temos abaixo, então, a soma dos sinais:



3 Convolução de Sinais

A ferramenta também realiza operações de convolução. Temos que a convolução e a soma das respostas ao impulso escalonadas pelo sinal de entrada. Podemos implementar facilmente essa soma com um laço. A fórmula da soma de convolução está abaixo:

$$\sum_{k=-\infty}^{+\infty} x[k]h[n-k] \tag{3}$$

Vamos mostrar a implementação da ferramenta na próxima seção.

3.1 Implementação da Ferramenta

A função convolve é a principal função da ferramenta. Ela realiza a convolução entre dois sinais de tempo discreto. O laço *for* realiza o processo da soma, como é possível ver. Foram determinados dois limites, para podermos iterar com mais segurança sobre a lista, e termos um resultado mais fidedigno. Caso o sinal se estenda demais é possível aumentá-los.

Veja a função abaixo:

```
INFERIOR_LIMIT = -100
SUPERIOR_LIMIT = 100

def convolve(x, h, n):
    y = 0
    for k in range(INFERIOR_LIMIT, SUPERIOR_LIMIT):
        y += x(k)*h(n - k)
    return y
```

3.2 Exemplos de Sinais

Os testes foram realizados em pares de sinais, onde um representa $x[n$ e o outro $h[n]$. O primeiro exemplo foi tirado da primeira prova da disciplina, questão 03, e os outros dois exemplos foram tirados do livro do Alan Oppenheim.

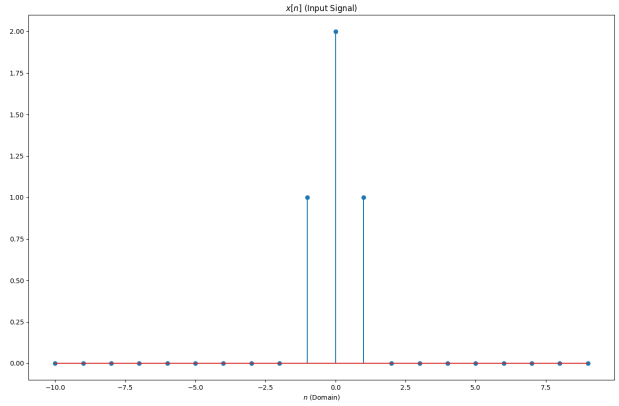
3.2.1 Par de Sinais da Terceira Questão da AP1

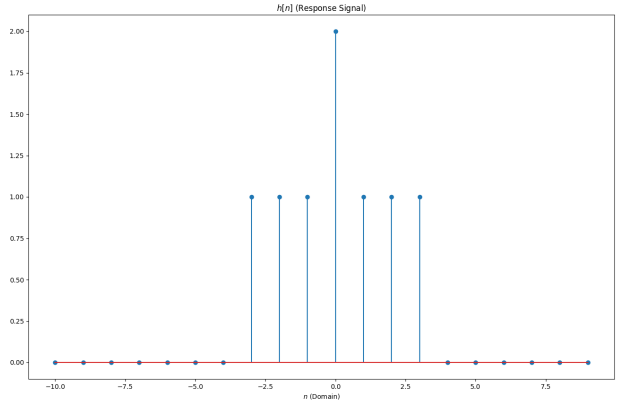
Esse sinal foi retirado da terceira questão da primeira prova da disciplina. Foi uma questão desafiadora, onde uma resposta ao impulso semelhante à entrada do sistema gera uma saída bastante interessante.

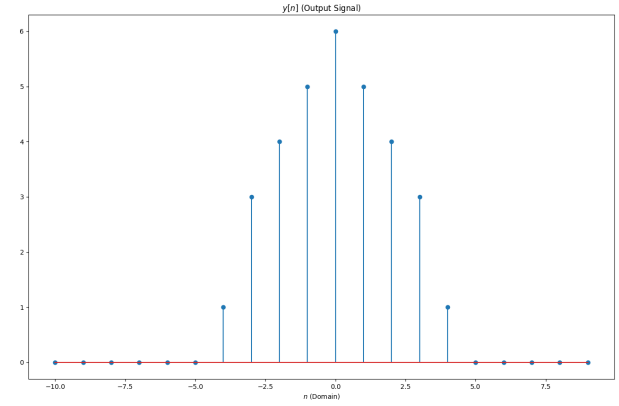
O código é semelhante aos códigos anteriores, onde temos funções que retornam um sinal, baseado no domínio que lhe é passado. O sinal de saída y[n] é o retorno da função de convolução da ferramenta, como é possível ver. Os três sinais são traçados e salvos em um arquivo ao final do programa.

```
import numpy as np
from convolution import *

def inputSignal(n):
    return 1*unitImpulse(n + 1) + 2*unitImpulse(n) + 1*unitImpulse(n - 1)

def responseSignal(n):
    return 1*unitImpulse(n + 3) + 1*unitImpulse(n + 2) + 1*unitImpulse(n + 1)
       + 2*unitImpulse(n) + 1*unitImpulse(n - 1) + 1*unitImpulse(n - 2) + 1*
       unitImpulse(n - 3)

n = np.arange(-10, 10)
y = convolve(inputSignal, responseSignal, n)
x = inputSignal(n)
h = responseSignal(n)

plotSignalAndSave(n, x, "$x[n]$ (Input Signal)", "images/first_pair_input.png"
    )
plotSignalAndSave(n, h, "$h[n]$ (Response Signal)", "images/
    first_pair_response.png")
plotSignalAndSave(n, y, "$y[n]$ (Output Signal)", "images/first_pair_output.
    png")
```

Podemos ver abaixo os gráficos gerados pela aplicação.





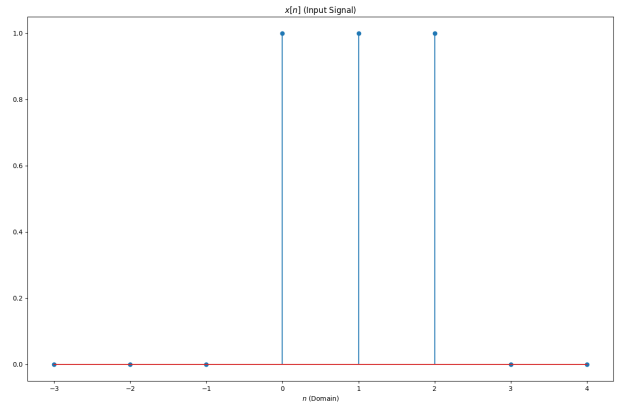


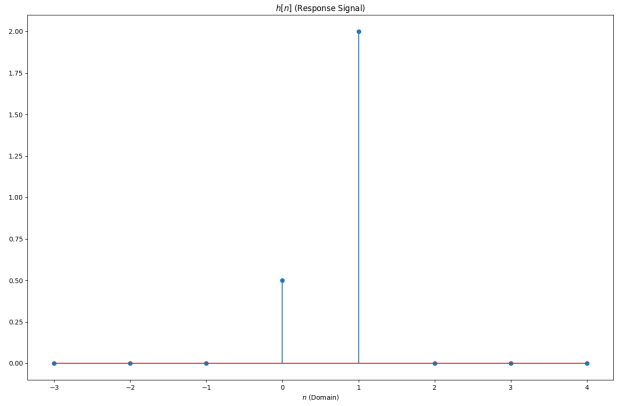
3.2.2 Par de Sinais do Exemplo 2.1

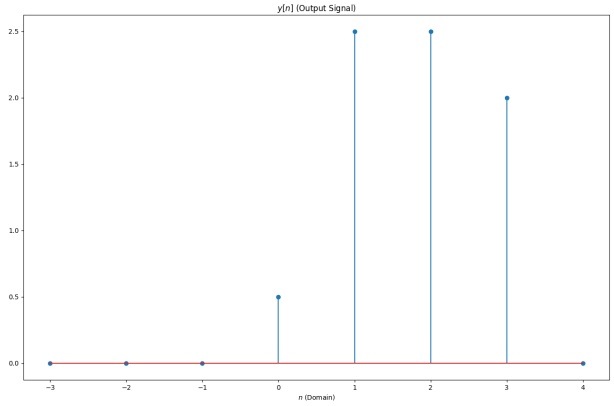
Os sinais desse código são bastantes simples também. Temos, ao final, os gráficos gerados pela aplicação.

```
import numpy as np
```

```
from convolution import *

def inputSignal(n):
    return 0.5*unitImpulse(n) + 2*unitImpulse(n - 1)

def responseSignal(n):
        return 1*unitImpulse(n) + 1*unitImpulse(n - 1) + 1*unitImpulse(n - 2)

n = np.arange(-3, 5)
y = convolve(inputSignal, responseSignal, n)
x = inputSignal(n)
h = responseSignal(n)

plotSignalAndSave(n, x, "$x[n]$ (Input Signal)", "images/second_pair_input.png
    ")
plotSignalAndSave(n, h, "$h[n]$ (Response Signal)", "images/
    second_pair_response.png")
plotSignalAndSave(n, y, "$y[n]$ (Output Signal)", "images/second_pair_output.
    png")
```







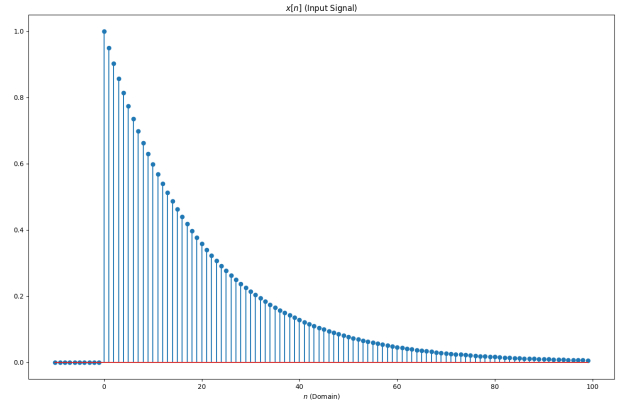
3.2.3 Par de Sinais do Exemplo 2.3

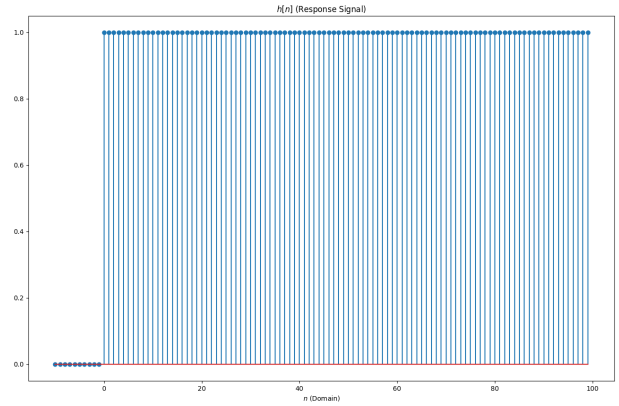
Esse código é um pouco mais complicado, pois os sinais utilizados aqui tem um grau de complexidade a mais. O domínio foi incrementado para que fosse possível visualizar todo o escopo do gráfico.

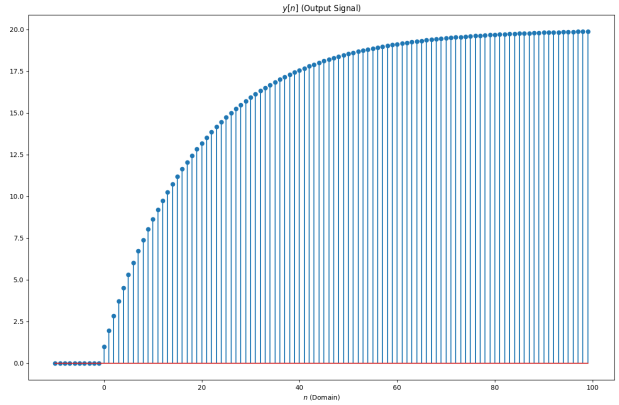
É possível perceber que o sinal de entrada, x[n], se aproxima de zero quando n se aproxima do infinito, e que o sinal de saída y[n] se aproxima de $\frac{1}{1-\alpha}$. Com $\alpha = 0.95$, temos que y[n] tende a 20. Podemos constatar isso apenas olhando para o gráfico.

```
import numpy as np
from convolution import *

ALPHA = 0.95

def inputSignal(n):
    return (ALPHA**n)*unitStep(n)

def responseSignal(n):
    return 1*unitStep(n)

n = np.arange(-10, 100)
y = convolve(inputSignal, responseSignal, n)
x = inputSignal(n)
h = responseSignal(n)

plotSignalAndSave(n, x, "$x[n]$ (Input Signal)", "images/third_pair_input.png"
    )
plotSignalAndSave(n, h, "$h[n]$ (Response Signal)", "images/
    third_pair_response.png")
plotSignalAndSave(n, y, "$y[n]$ (Output Signal)", "images/third_pair_output.
    png")
```

Podemos ver abaixo os gráficos gerados, usando uma escala aumentada.







4 Conclusão

O trabalho realmente agregou muito, nos trazendo conhecimentos práticos na usabilidade de ferramentas computacionais para resolver problemas relacionados a sinais de tempo discreto, conceito esse que é a base do processamento digital de sinais.

Ver os gráficos dos sinais sendo gerados pela linguagem foi uma experiência interessante, e acredito que gostaria de realizar uma disciplina de DSP.