UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Universidade Federal do Ceará - Campus Quixadá

Trabalho - AP 1
QXD0144 - Sinais e Sistemas

Prof. André Braga
28 de Janeiro de 2022

Aluno: Samuel Henrique Guimarães Alencar -
Matrícula: 473360

Quixadá-CE

Sumário

1 Introdução

O trabalho prático da cadeira de Sinais e Sistemas consiste na implementação de dois programas em *Python* capaz de fazer duas tarefas exigidas pelo professor.

A primeira tarefa exigida é que o primeiro programa seja capaz de fazer uma decomposição de sinal no tempo discreto, ou seja, separar o sinal em parte par e ímpar.

A segunda tarefa exigida é a realização de uma soma de convolução entre dois sinais de tempo discreto e finito.

OBSERVAÇÃO: não tente copiar e executar os códigos mostrados, pois nos trechos as bibliotecas não estão sendo importadas, então o algoritmo não irá rodar.

Para ter acesso aos códigos, basta clicar aqui para acessar o repositório no GitHub. Dentro do repositório tem as pastas referentes a cada parte do trabalho e dentro delas os *scripts* ou os arquivos do Notebook Jupyter que o GitHub é capaz de mostrar ao abri-los.

2 Decomposição de Sinais

De acordo com o livro-texto adotado na disciplina [2] existem duas fórmulas que são capazes de fazer essa decomposição. Essas fórmulas são:

$$x_{\text{par}}[n] = \frac{1}{2} \cdot (x[n] + x[-n])$$

$$x_{\text{ímpar}}[n] = \frac{1}{2} \cdot (x[n] - x[-n])$$

Sendo $x[n]$ o sinal de entrada. Perceba que ao somar a parte par com a parte ímpar o resultado é o próprio sinal de entrada, com isso, é possível ver que as fórmulas são matematicamente corretas.

2.1 Implementação

Para a implementação foi necessário o uso das seguintes bibliotecas: *numpy*, *pylab* e *math* para gerar gráficos e decompor os sinais.

2.1.1 Função expande domínio

```
def expansao(dom_sinal, n0, sinal):
    # Série numérica calculada entre o instante inicial até
    # o instante inicial mais o tamanho do array do sinal.
    domino = range(n0, n0 + len(sinal))

    #Um array de zeros com o tamanho do dominio do sinal de entrada
    sinal_expandido = np.zeros(len(dom_sinal))

    # É preciso verificar se este elemento
    # existe no domínio original através de index(n).
    # Se existir, devemos mapeá-lo
    # para o sinal expandido, caso contrário, a saída permanecerá zero.
    for i, n in enumerate(dom_sinal):
        try:
            sinal_expandido[i] = sinal[domino.index(n)]
        except ValueError:
            pass

    return sinal_expandido
```

A função acima foi implementada para que a operação de rebatimento do sinal pudesse acontecer sem que acontecesse erros. Pois, o rebatimento é como se fosse uma rotação do sinal em torno das ordenadas. Com isso, foi necessário expandir o domínio de modo que possuísse a mesma quantidade de números antes e após a origem. O sinal expandido é retornado ao final da função.

2.1.2 Função de decomposição do sinal

```
1  def decompor_sinal(sinal, n0):
2      #Calculando raio de intervalo para o domínio exendido
3      raio = max(abs(n0), abs(n0 + len(sinal) - 1))
4      dom_sinal = np.arange(-raio, raio+1)
5      sinal_entrada = expansao(dom_sinal, n0, sinal)
6      sinal_rebatido = sinal_entrada[::-1]
7
8      sinal_par = 0.5 * (sinal_entrada + sinal_rebatido)
9      sinal_impar = 0.5 * (sinal_entrada - sinal_rebatido)
10
11     return (dom_sinal, sinal_entrada, sinal_par, sinal_impar)
```

A função acima decompõe o sinal em parte ímpar e par. Mas antes, é calculado um raio de intervalo o valor máximo entre o ponto inicial (em módulo) e a soma do ponto inical com o tamanho do array -1 (em módulo). Com isso, o domínio do sinal é feito a partir da função de *numpy*, *arange*, um vetor uniformemente espaçado entre os valores passados com argumento é retornado. Após isso, o sinal de entrada é obtido com a chamada da função de expansão, passando o domínio, intervalo inicial e o sinal original como parâmetro, na linha 7.

Agora com o sinal estendido é possível fazer o rebatimento. Utilizando a notação de *slice*, é possível manipular um vetor ou lista. Então, a linha 6 irá retornar um vetor invertido, por exemplo, a = [1,2,3] se a operação b = a[::-1] for realizada, b será: b = [3,2,1]. Assim é feito o rebatimento do sinal.

Agora com a operação de rebatimento feita, basta realizar o cálculo das partes par e ímpar de acordo com as fórmulas mostradas no início da seção. Ao fim da função o domínio do sinal, o sinal de entrada e as partes par e ímpar são retornados para que o gráfico possar ser gerado.

2.1.3 Função para plotar o gráfico

```
def grafico_dividido(domino, original, par, impar):

    # Divide a janela de plotagem em quatro regiões.
    fig, ((g1,g2),(g3,g4)) = plt.subplots(2, 2, num=10)

    g1.stem(domino, original, "k-", "ko", "k-")
    g1.set_title("$x[n]$")
    g1.set_xlabel('n')

    g2.stem(domino, par, "k-", "ko", "k-")
    g2.set_title("$x_{par}[n]$")
    g2.set_xlabel('n')

    g3.stem(domino, impar, "k-", "ko", "k-")
    g3.set_title("$x_{ímpar}[n]$")
    g3.set_xlabel('n')

    g4.stem(domino, (impar + par), "k-", "ko", "k-")
    g4.set_title("$x_{ímpar}[n] + x_{par}[n]$")
    g4.set_xlabel('n')

    subplots_adjust(top=1.2,hspace=0.5)
    show()
```

Função feita para plotar os quatro gráficos em apenas uma imagem, então é necessário dividir a imagem em quatro plots de acordo com o trecho. Os gráficos a serem plotados são: gráfico original, gráfico par, gráfico ímpar e a soma do sinal par e ímpar. Os argumentos "k-", "ko"e "k-"são utilizados para definir o traço, o ponto e cor de ambos. O primeiro "k-"significa uma linha (-) de cor preta (k) no "eixo Y", o "ko"significa um ponto preto (k) na ponta de linha e o último "k-"é a linha (-) preta no "eixo X".

Além dessa função, também existe a função que imprime cada sinal separadamente, clique aqui para ir ao repositório.

2.2 Testes e gráficos gerados

2.2.1 Sinal ímpar: $x[n] = \sin\left(\frac{n\pi}{2}\right)$

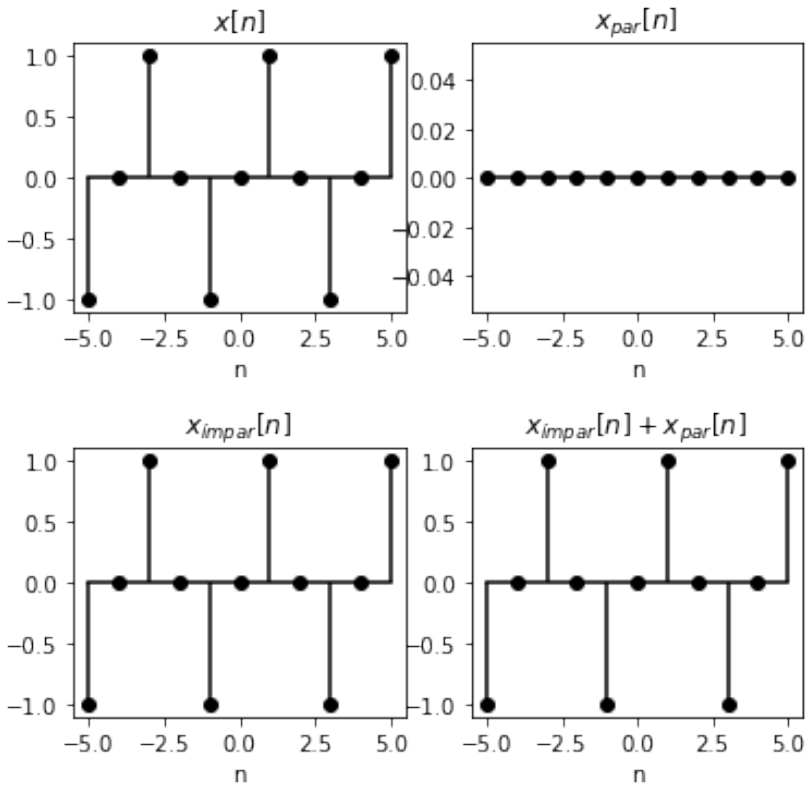

Figura 1: $x[n] = \sin\left(\frac{n\pi}{2}\right)$

2.2.2 Sinal par: $x[n] = e^{\cos(n\pi)}$

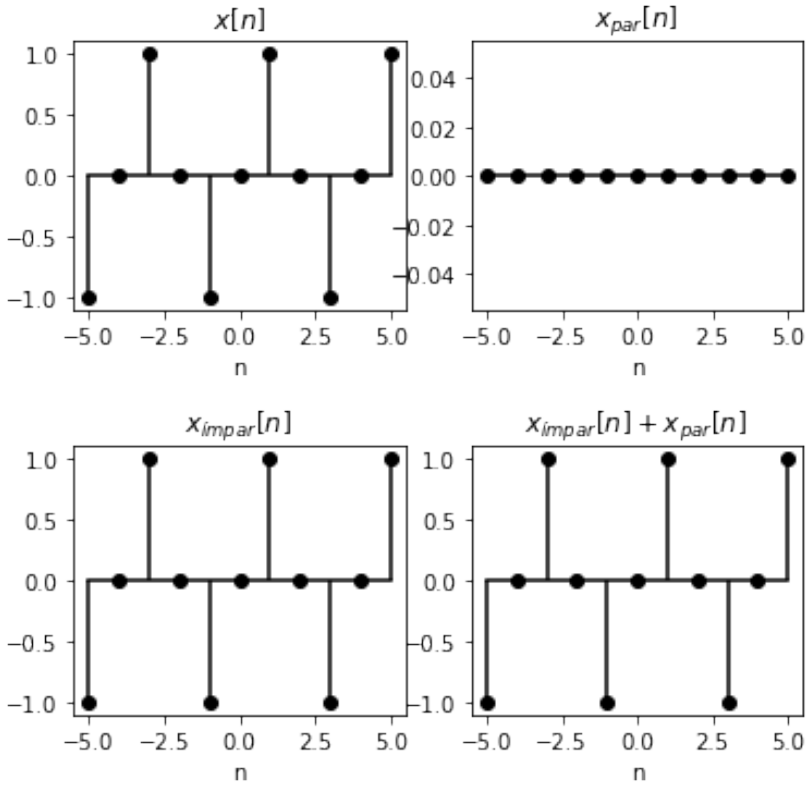

Figura 2: $x[n] = e^{\cos(n\pi)}$

2.2.3 Sinal par: $x[n] = \sin\left(\frac{n\pi}{2}\right) + e^{\cos(n\pi)}$

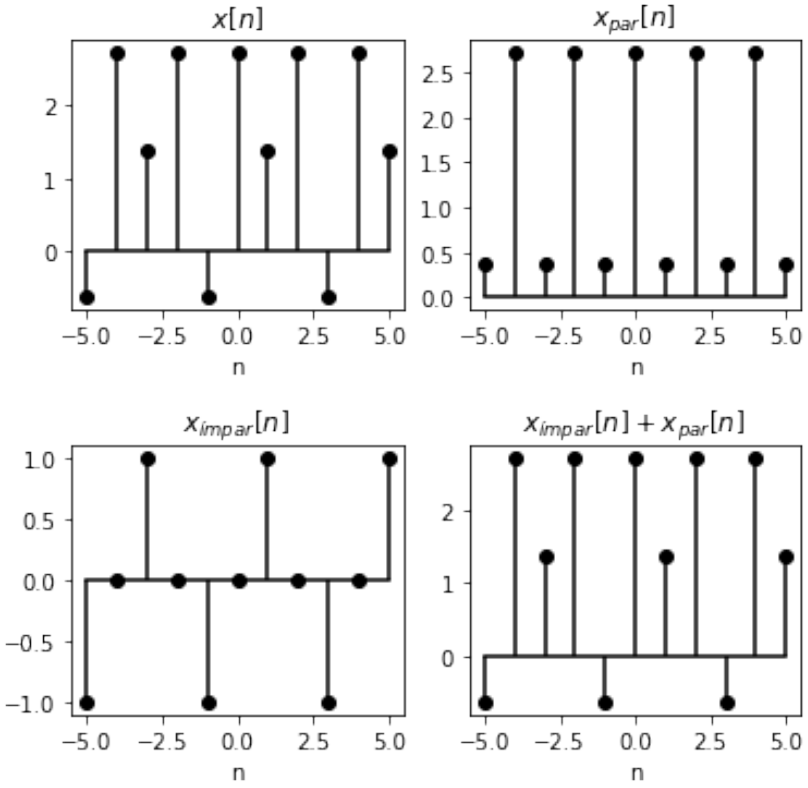

Figura 3: $x[n] = \sin\left(\frac{n\pi}{2}\right) + e^{\cos(n\pi)}$

3 Convolução de sinais

A segunda parte do trabalho consiste em implementar um algoritmo capaz de realizar a convolução entre dois sinais de tempo discreto.

A convolução entre dois sinais pode ser dada pela seguinte equação: [2]

$$x[n] * h[n] = y[n] = \sum_{k=-\infty}^{\infty} x[k] \cdot h[n-k]$$

Onde: $x[n]$ corresponde ao sinal de entrada, $h[n]$ a resposta ao impulso e $y[n]$ o sinal de saída ou o resultado da convolução.

3.1 Implementação

Assim como na primeira parte, o *numpy* e o *pylab* foram utilizados.

3.1.1 Função de impulso unitário (δ)

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

Para contextualizar, a função *where* da biblioteca *numpy* tem a seguinte estrutura: where(condição, [x, y]). A função checa a condição, se ela for verdadeira, então um *array* de valores x é retornado. Se a condição for falsa, um *array* de valores y é retornado.

Então, no código em questão o intervalo n passado como argumento é verificado, se ele for igual a zero, um *array* de 1's é retornado ou um *array* de 0's, caso contrário.

3.1.2 Função para plotar o gráfico

```
def grafico_divido(dominio, impulso, entrada, resp_impulso, sinal_conv):
    fig, ((g1,g2),(g3,g4)) = plt.subplots(2, 2, num=10)

    g1.stem(dominio, impulso, "k-", "ko", "k-")
    g1.set_title("$\delta[n]$")
    g1.set_xlabel('n')

    g2.stem(dominio, entrada, "k-", "ko", "k-")
    g2.set_title("$x[n]$")
    g2.set_xlabel('n')

    g3.stem(dominio, resp_impulso, "k-", "ko", "k-")
    g3.set_title("$h[n]$")
    g3.set_xlabel('n')

    g4.stem(dominio, sinal_conv, "k-", "ko", "k-")
    g4.set_title("$y[n]$")
    g4.set_xlabel('n')

    subplots_adjust(top=1.2,hspace=0.5)
    show()
```

Seguindo o mesmo esquema da função da primeira parte do trabalho, a figura é dividida em 4 e o impulso, sinal de entrada, resposta ao impulso e sinal de saída são plotados. Os argumentos "k-", "ko"e "k-"são utilizados para definir o traço, o ponto e cor de ambos. O primeiro "k-"significa uma linha (-) de cor preta (k) no "eixo Y", o "ko"significa um ponto preto (k) na ponta de linha e o último "k-"é a linha (-) preta no "eixo X".

Além dessa função, também existe a função que imprime cada sinal separadamente, clique aqui para ir ao repositório.

3.2 Testes e gráficos gerados

3.2.1 Sinal 1

Impulso unitário: $\delta[n] = \begin{cases} 1, & n = 0 \\ 0, & n \neq 0 \end{cases}$

Entrada: $x[n] = \delta[n] + \delta[n-1] + \delta[n-2]$

Resposta ao impulso: $h[n] = 3 \cdot \delta[n] + 2 \cdot \delta[n-1] + \delta[n-2]$

```
1  #Sinal de entrada
2  def entrada(n):
3      return impulso(n) + impulso(n-1) + impulso(n-2)
4
5  #Resposta ao impulso h[n]
6  def h(n):
7      return 3*impulso(n) + 2*impulso(n-1) + impulso(n-2)
8  n = np.arange(-3,7) # Intervalo -3 <= n < 7
9
10 # Resposta da convolução
11 y = entrada(0)*h(n) + entrada(1)*h(n-1) + entrada(2)*h(n-2)
12
13 x = entrada(n)
14 resp_imp = h(n)
15 delta = impulso(n)
16 grafico_divido(n, delta, x, resp_imp, y)
```

A trecho acima realiza o cálculo da soma de convolução. Na linha 2 temos a definição do sinal de entrada que recebe como argumento um ou mais intervalos de tempo discreto, a função faz chamada da função de impulso vista anteriormente.

Com isso, o sinal de entrada é um sinal com 3 pontos de valor 1 no "eixo Y"e nos instantes 0, 1 e 2. Já a resposta ao impulso ($h[n]$) seria um sinal decrescente também nos instantes 0, 1 e 2. No instante 0 o valor 3, no instante 1 o valor 2 e por fim, no instante 0 o valor 1.

Com os sinais $x[n]$ e $h[n]$ definidos, agora basta seguir a fórmula mostrada no início da seção. Na linha 11 do trecho a soma de convolução é realizada. A seguir o gráfico gerado pelo algoritmo é apresentado.

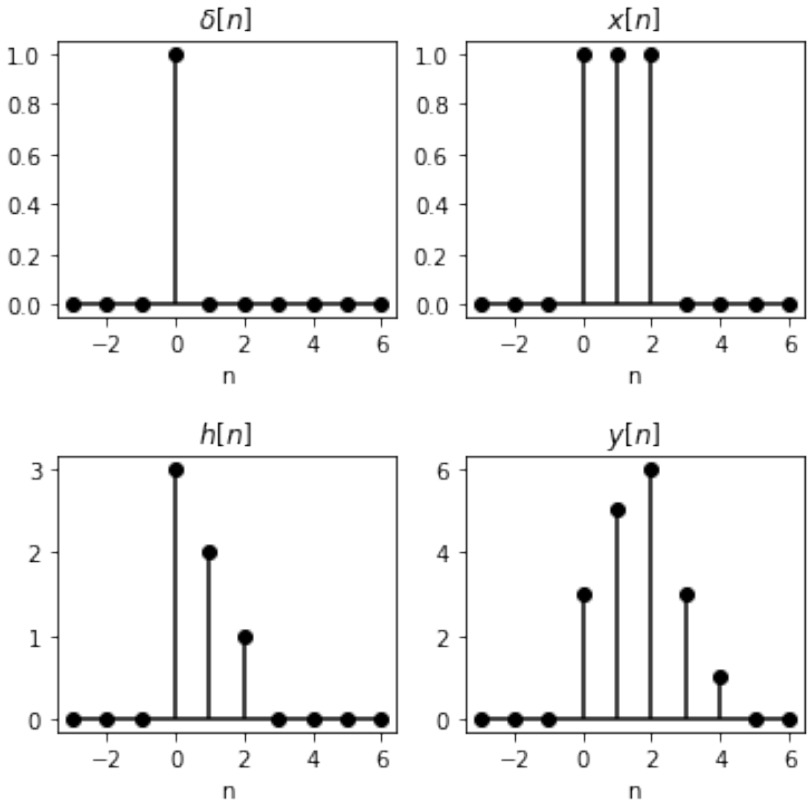

Figura 4: Resultado do teste 1

3.2.2 Sinal 2 - Exemplo do livro [2]

O exemplo 2.1 na página 51 tem como entrada e resposta ao impulso:

Impulso unitário: $\delta[n] = \begin{cases} 1, & n = 0 \\ 0, & n \neq 0 \end{cases}$

Entrada: $x[n] = 0.5 \cdot \delta[n] + 2 \cdot \delta[n-1]$

Resposta ao impulso: $h[n] = \delta[n] + \delta[n-1] + \delta[n-2]$

```
1  #Sinal de entrada
2  def entrada(n):
3      return 0.5*impulso(n) + 2*impulso(n-1)
4
5  #Resposta ao impulso h[n]
6  def h(n):
7      return impulso(n) + impulso(n-1) + impulso(n-2)
8
9  n = np.arange(-2, 5) # Intervalo -1 <= n < 6
10
11 # Resposta da convolução
12 y = entrada(0)*h(n) + entrada(1)*h(n-1)
13
14 x = entrada(n)
15 resp_imp = h(n)
16 delta = impulso(n)
17 grafico_divido(n, delta, x, resp_imp, y)
```

Seguindo o mesmo princípio do primeiro exemplo, no trecho acima temos a definição do sinal de entrada, da resposta ao impulso e da soma de convolução. Na linha 11 a convolução é feita seguindo a fórmula que foi apresentada. A seguir o gráfico gerado pelo algoritmo é apresentado.

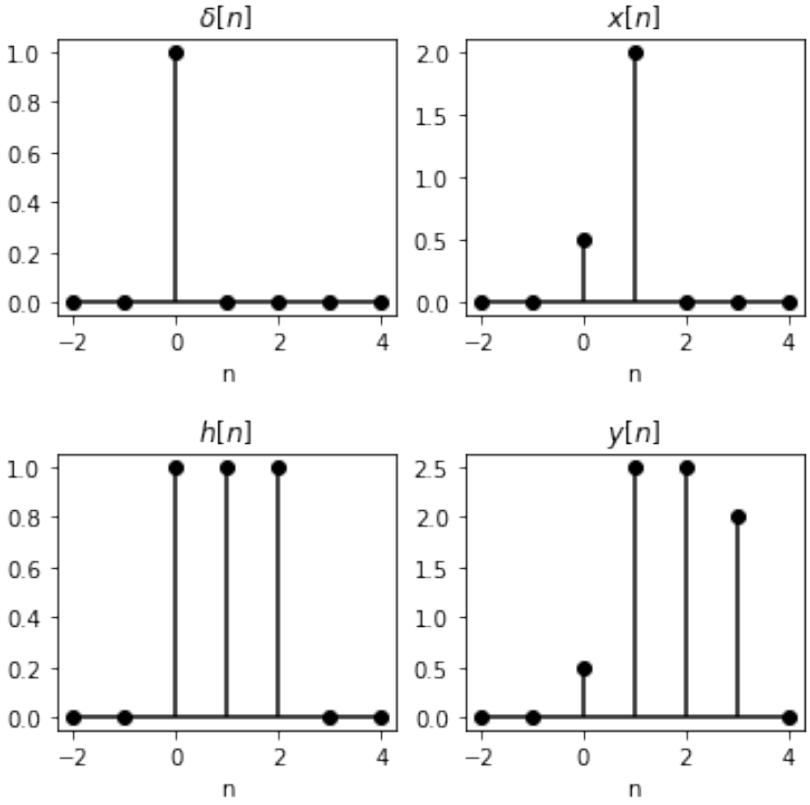

Figura 5: Resultado do teste 2

4 Conclusão

O trabalho foi muito agregador, sempre tive curiosidade em como esses gráficos no tempo discretos eram gerados na linguagem em questão.

Além disso, ao fazer esse trabalho foi possível solidificar os conhecimentos em relação aos assuntos requisitados.

O trabalho exigiu um conhecimento que eu ainda não tinha sobre a linguagem, pois só sabia o básico do básico, então foi necessário entender melhor como a linguagem funcionava em algumas partes do trabalho e pesquisar algumas funções que ainda não estavam claras, acredito que essa tenha sido a parte mais difícil e demorada.

Referências

[1] Introdução ao processamento digital de sinais com numpy. `http://ipds.wikidot.com/script:signal-compression`. Acessado em: 18/01/2022.

[2] A.V. Oppenheim. *Sinais e sistemas*. Prentice-Hall, 2010.