**Projeto 3 - Códigos**

Nesse projeto, utilizamos a definição de diversas funções para os cálculos desejados. Abaixo estão listadas todas as funções e rotinas utilizadas no código.

```
1
2     from typing import Type
3     import numpy as np
4     import matplotlib.pyplot as plt
5     import pandas as pd
6     import seaborn as sb
7     from seaborn.palettes import color_palette
8
9
```

Listing 1: Importação das bibliotecas

# 1 Matrizes de dados para a criação dos autômatos

Para a geração de cada tipo de autômato, utilizamos as seguintes matrizes:

```
1
2     sawTooth = np.matrix([[0.5, 0, 0, 0.7], [0.5, 0.1, 0, 0], [0, 0.9, 0.6,
      0], [0, 0, 0.4, 0.3]])
3
4     triangular = np.matrix([[0.1, 0, 0, 0, 0, 0.9], [0.9, 0.2, 0, 0, 0, 0],
      [0, 0.8, 0.1, 0, 0, 0], [0, 0, 0.9, 0.3, 0, 0], [0, 0, 0, 0.7, 0.2, 0],
      [0, 0, 0, 0, 0.8, 0.1]])
5
6     rectangular = np.matrix([[0.9, 0.1], [0.1, 0.9]])
7
8     matProbs = [sawTooth, rectangular, triangular]
9
10
```

Listing 2: Matrizes estocásticas e matriz que agrupa todas as demais

# 2 Geração dos autômatos

Para a geração de um autômato, definimos a função a seguir que aceita o número de passos do autômato, a matriz estocástica e o tipo de onda que queremos gerar. Esse tipo de onda é utilizado apenas para diferenciar a formação de ondas dente de serra e triangulares.

```
1
2     def genericAutomato(n, mat, type="general", mat2=0): #mat must be a
    numpy mXm array
3     '''
4     Type can be:
5     "general"
6     "triangular"
7
8     '''
9
10    automata = np.zeros(n)
11    automata[0] = 1 # we must start at the node zero
12    shape = mat.shape
13
14    ##Is it a squared matrix?
15    if (shape[0] == shape[1]):
16        nodes = shape[0]
17    else:
18        print(shape[0])
19        print(shape[1])
20        print("Not a squared matrix")
21        return 1
22
23
24    if (type=="general"):
25        ##Building the signal based on the stochastic matrix
26        for i in range(1, n): #n steps on the automato
27            value = np.random.rand(1)
28            current = int(automata[i-1])
29            probs = mat[:, current]
30
31            sumPrb = 0
32
33            for j in range(nodes):
34
35                sumPrb = sumPrb + probs[j]
36
37                if(sumPrb >= value):
38
39                    automata[i] = j
40
41                    break
42
43    if(type=="triangular"):
44        for i in range(1, n): #n steps on the automato
45            value = np.random.rand(1)
46            current = int(automata[i-1])
47            probs = mat[:, current]
```

```
            sumPrb = 0

            for j in range(nodes):

                sumPrb = sumPrb + probs[j]

                if(sumPrb >= value):

                    automata[i] = j

                    break
        automata = np.where(automata==4, 2, automata)
        automata = np.where(automata==5, 1, automata)


    return automata


```

Listing 3: Geração de autômatos

Para a implementação do autômato triangular, utilizamos uma matriz com 5 nós e na geração do autômato, os valores 4 e 5 aparecem. Para que tais valores sejam transformados em 2 e 1, respectivamente, utilizamos a modificação de elementos do vetor *where* da biblioteca numpy.

# 3 Derivada e Integral

Funções para transformação do conjunto de dados do vetor autômato.

```

    def derivative(vector):

        derivative = np.zeros(len(vector))
        for i in range(1, len(vector)):
            derivative[i] = vector[i] - vector[i-1]

        return(derivative)


```

Listing 4: Derivada

Vale notar que utilizamos $\Delta t = 1$ para os cálculos de derivada.

```
    def integration(vector):
        integration = np.zeros(len(vector))
        sum = 0
        for i in range(len(vector)):
            integration[i] = sum
            sum+=vector[i]
        return(integration[-1])


```

Listing 5: Integral

Note que a rotina nos da integração nos retorna o último valor do vetor construído que representa a área embaixo da curva. Esse parâmetro é um dos que utilizamos em nossa tentativa de separação dos conjuntos.

# 4 Entropia e paridade

Para extração dos parâmetros relacionados à entropia de Shannon, implementamos a seguinte função que calcula a entropia e a paridade do autômato.

```
    def s(vector):
        s = 0
        values, counts = np.unique(vector, return_counts=True)
        probs = np.zeros(len(values))


        for i in range(len(values)):
            probs[i] = counts[i]/len(vector)

        for i in range(len(probs)):
            s+= probs[i]*np.log2(probs[i])

        return([-s, 2**(-s)])


```

Listing 6: Entropia e paridade

# 5 Distância entre símbolos

Na extração do parâmetro distância entre símbolos, implementamos a função a seguir que calcula e constrói um vetor com a distancia média para todos os possíveis símbolos do autômato

e nos retorna a média desse vetor. Em outras palavras, calculamos a média das médias das distâncias para todos os símbolos.

```
def intersymbleDistance(vector):
    values= np.unique(vector)
    distances = []
    idist = []
    for i in range(len(values)):
        symbol = values[i]
        for j in range(len(vector)-1):
            if(vector[j] == symbol):
                d0 = j
                for k in range(j+1, len(vector)):
                    if(vector[k] == symbol):
                        d1 = k
                        j=k
                        break
                idist.append(d1-d0)
        distances.append(np.mean(idist))
    return np.mean(distances)
```

Listing 7: Distância entre símbolos

# 6 Tamanho dos *Bursts*

Ainda sobre parâmetros relacionados aos símbolos do autômato, temos o cálculo do tamanho médio dos *bursts* para todos os símbolos existentes no autômato.

```
def burstSize(vector):
    ##returns the mean of all symbols burst sizes
    values= np.unique(vector)
    sizes = []

    for i in range(len(values)):

        symbol = values[i]

        for j in range(len(vector)-1):

            current = vector[j]

            if(current == symbol):

```

```
                    length = 0

                    while(current == symbol and j<(len(vector)-1)):

                        length+=1
                        j+=1
                        current = vector[j]

                    j=length
                    sizes.append(length)

        return np.mean(sizes)

```

Listing 8: Tamanho dos Bursts

# 7 Transformação de espaço

Utilizando a transformada de Fourier, pudemos calcular o espectro de potencia de nosso autômato a partir da multiplicação do vetor transformado (já no espaço de frequências) com seu complexo conjugado. Essa multiplicação nos retorna um vetor e extraímos o valor da frequência cuja a intensidade era a maior nesse novo vetor como nosso parâmetro.

```

    def powerSpectrumMaxFreq(vector):
        fft = np.fft.fft(vector)
        fft = np.delete(fft, 0)[0:int(len(fft)/2)]
        conj = np.conjugate(fft)
        #freq = np.fft.fftfreq(fft.size)
        pw = fft*conj
        """ plt.stem(pw)
        plt.xlabel("Freq")
        plt.ylabel("Intencidade")
        plt.show()
        input() """
        return(np.argmax(pw))


```

Listing 9: Espectro de potencia

# 8 Grafos

Passando para a representação de nosso autômato como um grafo, implementamos o pseudo-código da visibilidade apresentado no CDT-23 para a extração de dois parâmetros: Número de

conexões totais de nosso grafo e a razão entre o número de conexão realizadas e as possíveis conexões.

```
    def visibility(vector):

        a = np.zeros((len(vector), len(vector)))
        for j in range(2, len(vector)):
            for i in range(1,j-1):
                flag = 1
                k = i+1
                while(k<=j-1 and flag ==1):
                    aux = vector[j]+(vector[i]-vector[j])*(j-k)/(j-i)
                    if(vector[k]>=aux):
                        flag=0
                    k = k+1
                if(flag == 1):
                    a[i, j] = 1
                    a[j, i] = 1
        values, ocorrence = np.unique(a, return_counts=True)

        link = ocorrence[1]/ocorrence[0]

        nLinks = ocorrence[1]
        """ plt.matshow(a, cmap="plasma")
        plt.show()
        input() """
        return(nLinks, link)


```

Listing 10: Rotina de visibilidade

# 9 Construção dos gráficos 2 a 2

Após a implementação de todas as funções que calculam os parâmetros definidos, implementamos uma ultima que é responsável pela geração de 50 autômato dos três tipos desejados, calcula todos os parâmetros para cada um dos autômatos agrupa todos os dados em uma estrutura da biblioteca *Pandas* (dataframe) para organização e facilitação do plot. A partir dessa estrutura, para o plot 2 a 2 dos parâmetros, utilizamos a biblioteca *seaborn*.

```

    def by2(matProbs):
        columns = ["Type", "stdRelFrq","Ent", "Even", "symbolDist", "
    BurstSize", "PSpecMaxFreq", "#Links", "link/possible", "Area"]

```

```
5
6       #columns = ["Entropy", "Eveness", "IntersymbolDistance"]
7       t = ["sawTooth", "rectangular", "triangular"]
8
9
10
11      auto = []
12      nameIndex = 0
13      for kind in matProbs:
14
15          name = t[nameIndex]
16
17          if(name == "triangular"):
18              ty = "triangular"
19          else:
20              ty = "general"
21
22          for i in range(50):
23
24              a = genericAutomato(200, kind, type=ty)
25              entropy = s(a)
26              ent = entropy[0]
27              even = entropy[1]
28              interS = intersymbleDistance(a)
29              burst = burstSize(a)
30              pwMF = powerSpectrumMaxFreq(a)
31              nodes = visibility(a)
32              nLinks, nodeMean = nodes[0], nodes[1]
33              std = relativeFrequency(a)[1]
34              area = integration(a)
35              auto.append([name, std, ent, even, interS, burst, pwMF,
    nLinks, nodeMean, area])
36              #auto.append([ent, even, interS])
37          nameIndex+=1
38
39      data = pd.DataFrame(auto, columns=columns)
40      print(data)
41      sb.pairplot(data, hue="Type", diag_kind="None")
42      plt.show()
43
```

Listing 11: Construção dos gráficos 2 a 2