

UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO



**TRANSFORMANDO SÉRIES TEMPORAIS EM COMPLEXOS
SIMPLICIAIS**

Discente: Laura Alves Pacifico da Silva

Professor: Fernando Jose Castor De Lima Filho

Recife

2020

1 Introdução

Em topological data analysis (TDA) estudamos os dados olhando para forma destes, é uma abordagem para a análise de conjuntos de dados usando técnicas de topologia algébrica. A principal ferramenta é a homologia persistente, uma adaptação da homologia para dados. Originalmente os dados que estudo são apresentados na forma de séries temporais (mais especificamente séries temporais financeiras, por sua natureza caótica), no entanto a entrada para TDA é um point cloud, então precisamos transformar a série temporal em uma nuvem de pontos preservando todas as características topológicas que são importantes nessa série (para isso garantimos a existência de um difeomorfismo). Após esse procedimento “triangularizamos” essa superfície, transformando-a em um complexo simplicial, este representa perfeitamente o point cloud e nos permite extrair informações relevantes sobre tal conjunto. Tais como números de Betti e Características de Euler dentre outros.

2 Objetivo

Neste projeto implementamos a conversão da série temporal em complexos simpliciais de maneira fácil, pois este conta com uma interface. A série é inicialmente (fornecida pelo usuário) em uma nuvem de pontos (fazendo um embedding 1D, 2D, 3D ou 4D) e posterior a isso é contruída a matriz de distâncias e plotado o ultimo complexo simplicial totalmente conexo, este “representa” a nuvem de pontos que foi gerada no passo anterior. Possibilitando assim a extração de informações topológicas essenciais no estudo via TDA.

No desenvolvimento foram utilizadas as bibliotecas Matplotlib, Numpy, tkinter, e networkx.

Uma versão deste projeto pode ser encontrada também no Google Colab, no entanto esse não conta com a interface (por limitações de sua natureza), e sim com entradas pelo teclado.

Link para acesso:
<https://colab.research.google.com/drive/1v0w2zKuN9SUrTO8s5EsSw0MX-CqSTMei?usp=sharing>

4 Funcionalidades

O programa implementado possui as seguintes funcionalidades:

1. Através de interface, colhe as informações fornecidas pelo usuário: Série temporal e dois parâmetros para reconstrução;
2. Plota a série temporal;
3. Obtém a reconstrução do espaço, de acordo com os parâmetros informados;
4. Plota a reconstrução de acordo com a dimensão informada;
5. Calcula a matriz de distâncias (útil quando vamos calcular características topológicas);
6. Plota o "complexo" completamente conectado.

4 O Algoritmo

Inicialmente fornecemos a o usuário um plot da série que foi fornecida, utilizando a biblioteca do matplotlib.

```
# plota a série fornecida
def plotSerie(serief):
    plt.plot(serief, color='blue')
    plt.title('Time Series')
    plt.xlabel('x')
    plt.ylabel('f(x)')
    # plt.axis('off') caso não queria que os eixos apareçam
```

De posse da série temporal a convertemos em um point cloud através de sliding window, onde são utilizados os parâmetros d e τ fornecidos pelo usuário.

```
def slidingWindows(d, t, series):
    coords = []
    for i in range(0, len(series) - 1):
        if i + d * t > len(series) - 1:
            return coords
        else:
            point = []
            for j in range(0, d + 1):
                point.append(series[i + j * t])
            coords.append(point)
    return coords
```

Após a construção da point cloud (ou window, como denotei) são construídos os gráficos de dispersão que o representam. Estes foram construídos com matplotlib usando scatter plot, aqui estão representadas as versões 2D, 3D e 4D (onde é acrescentado um mapa de calor para representar a 4 dimensão, para isso também utilizei o numpy).

```
def plot2d(window):
    fig = plt.figure()

    ax = fig.add_subplot(111)
    x = []
```

```

y = []
for i in window:
    x.append(i[0])
    y.append(i[1])

ax.scatter(x, y, s=30, c='r')
ax.text(0.05, 0.95, "", transform=ax.transAxes)

plt.show()

def plot3d(window):
    fig = plt.figure()

    ax = fig.add_subplot(111, projection='3d')
    x = []
    y = []
    z = []
    for i in window:
        x.append(i[0])
        y.append(i[1])
        z.append(i[2])
    print(len(x), len(y), len(z))

    ax.scatter3D(x, y, z, s=30, c='r')
    ax.text2D(0.05, 0.95, "", transform=ax.transAxes)

    plt.show()

def plot4d(window):
    fig = plt.figure()

    ax = fig.add_subplot(111, projection='3d')
    x = []
    y = []
    z = []
    t = []
    for i in window:
        x.append(i[0])
        y.append(i[1])
        z.append(i[2])
        t.append(i[3])
    x2 = np.array(x)
    y2 = np.array(y)
    z2 = np.array(z)
    c = np.array(t)
    img = ax.scatter(x2, y2, z2, c=c, cmap=plt.hot())
    fig.colorbar(img)
    plt.show()

```

Após o plot da nuvem de pontos iniciaremos uma sequência de funções com o objetivo de calcular a matriz de distâncias e por fim fazer o plot do complexo simplicial. Para isso começamos convertendo a window num array. Utilizamos `np.unique` afim de obter o `np_array` exclusivo (sem repetições). Por exemplo:

```
a = np.array([[1, 0, 0], [1, 0, 0], [2, 3, 4]])
np.unique(a, axis=0)
array([[1, 0, 0], [2, 3, 4]])
```

Assim,

```
##Converter window num numpy array. Retorna unique values (esses
valores foram plotados).
def toNumpyArray(window): # p encontrar a matriz de dists
    np_array = np.unique(np.asarray(window), axis=0)
    return np_array
```

De posse do `np_array` obtido obtemos as distancias para todos os pontos, iniciamos com uma lista vazia e dentro dela uma linha (que também é uma lista vazia) e a medida que vamos calculando a distancia vamos adicionando a linha e posteriormente a coluna. As distâncias são calculadas com a função `np.linalg.norm()` que calcula a norma matricial (vetorial) de acordo com os parâmetros fornecidos.

```
# Obter a Matriz de Distância para/de todos os pontos
def getDistanceMatrix(numpy_window):
    dist_matrix = []
    for i in numpy_window:
        line = []
        for j in numpy_window:
            dist = np.linalg.norm(i - j)
            line.append(dist)
        dist_matrix.append(line)
    return np.asarray(dist_matrix)
```

Com a matriz de distâncias calculada, criamos uma matriz vazia (com a mesma quantidade de linhas e colunas da matriz de distâncias) para começarmos a criar os complexos simpliciais.

```
## Matriz simplicial
def getEmptySimplicialMatrix(dist_matrix):
    simplicial_matrix = np.zeros((len(dist_matrix[0]),
len(dist_matrix)))
    return simplicial_matrix
```

A função `updateSimplicialMatrix` irá verificar quando duas arestas devem se conectar, em outras palavras ele vai sempre atualizar a `getSimplicialList` da seguinte maneira: se 2 pontos são menores ou iguais a distância, atualize o valor na matriz simplicial para 1.

```
##update na matriz simplicial. Critério : (se 2 pontos são menores ou
iguais a distancia, atualize o valor na matriz simplicial para 1)
def updateSimplicialMatrix(simplicial_matrix, dist_matrix, dist):
    for i in range(0, len(dist_matrix)):
        for j in range(0, len(dist_matrix)):
            if dist_matrix[i][j] <= dist:
                simplicial_matrix[i][j] = 1
    return simplicial_matrix
```

Assim esse processo é repetido até todos os elementos estejam completamente conectados.

```
##obter lista de simplices atualizando a distância de início até que
todos os "elementos" estejam conectados
def getSimplicialList(simplicial_matrix, dist_matrix, start_dist,
step):
    simplicial_list = []
    while np.all(simplicial_matrix) != 1:
        simplicial_matrix = updateSimplicialMatrix(simplicial_matrix,
dist_matrix, start_dist)
        simplicial_list.append(simplicial_matrix.copy())
        start_dist = start_dist + step
    return simplicial_list
```

Como queremos visualizar esse complexo, retiramos repetições.

```
##Método auxiliar para obter apenas os simpliciais
def getUniqueSimplicialList(simplicial_list):
    global q
    unique_simplicial_list = [simplicial_list[0]]
    for simplicial in simplicial_list:
        for unique_simplicial in unique_simplicial_list:
            q = 0
            if (simplicial == unique_simplicial).all():
                q = 1
                break
        if q == 0:
            unique_simplicial_list.append(simplicial)
    return unique_simplicial_list
```

Agora, como queremos visualizar o complexo completamente conectado, vamos “transformar” nossa lista em tuplas para podermos plotar o grafo.

```
## Todos as tuplas "válidas" desde simplices até criar os grafos
def getTuplesFromSimplicialList(simplicial_list):
    list_of_tuples = []
    for simplicial in simplicial_list:
        tuples = []
        for i in range(0, len(simplicial)):
            for j in range(0, len(simplicial[i])):
                if simplicial[i][j] == 1:
                    tuples.append((i, j))
        list_of_tuples.append(tuples)
    return list_of_tuples
```

Agora, construímos o grafo utilizando a biblioteca networkx.

```
## Criando os grafos a partir das tuplas de simpliciais
def getGraphList(list_of_tuples):
    graph_list = []
    for i in list_of_tuples:
        g = nx.Graph()
        g.add_edges_from(i)
        graph_list.append(g)
    return graph_list
```

E por fim, plotamos o grafo completamente conectado que corresponde ao ultimo complexo simplicial.

```
##Plot grafo
options = {
    'node_color': 'red',
    'node_size': 50,
    'width': 0.1,
}

def plotGraph(graph):
    nx.draw(graph, **options)
    plt.show()
```

Agora, já com as funções auxiliares implementadas, foi construída uma interface usando o Tkinter. Inicialmente foi criada uma classe chamada “Interface” e nela os controles que são exibidos na tela. Foram criados containers que são as estruturas que armazenam os widgets. Após a criação dos containers posicionamos os elementos na tela com o gerenciador geométrico pack() e foi criado um widgetLabel para imprimir o texto. Para receber as informações do usuário usamos o widgetEntry. Para executar a ação criamos um widgetBottom que recebe como comando `converteTsinSC()` que executa a sequência de funções auxiliares já implementadas acima. Por fim instanciamos a classe Tk() através da variável janela (assim permitindo a utilização dos widgets). Passamos a variável janela como parâmetro do construtor da classe Interface e chamamos `janela.mainloop()` para exibirmos a tela.

```
class Interface:
    def __init__(self, master=None):
        self.fontePadrao = ("Times", "10")
        self.primeiroContainer = Frame(master)
        self.primeiroContainer["pady"] = 20
        self.primeiroContainer.pack()

        self.segundoContainer = Frame(master)
        self.segundoContainer.pack()

        self.terceiroContainer = Frame(master)
        self.terceiroContainer["pady"] = 20
        self.terceiroContainer["padx"] = 20
        self.terceiroContainer.pack()

        self.quartoContainer = Frame(master)
        self.quartoContainer["pady"] = 20
        self.quartoContainer["padx"] = 20
        self.quartoContainer.pack()

        self.quintoContainer = Frame(master)
        self.quintoContainer["pady"] = 20
        self.quintoContainer["padx"] = 20
        self.quintoContainer.pack()

        self.titulo = Label(self.primeiroContainer, text="Convertendo
Séries Temporais em Complexos Simpliciais",
                             fg="purple")
        self.titulo["font"] = ("Helvetica", "16", "bold")
        self.titulo.pack()
```

```

        self.tipoEntraLabel = Label(self.segundoContainer,
text="Informe a Série Temporal:", font=self.fontePadrao)
        self.tipoEntraLabel.pack(side=LEFT)

        self.tipoEntra = Entry(self.segundoContainer)
        self.tipoEntra["width"] = 81
        self.tipoEntra["font"] = self.fontePadrao
        self.tipoEntra.pack(side=LEFT)

        self.dimensaoLabel = Label(self.terceiroContainer,
                                text="Escolha o parâmetro d,
lembre-se que ele corresponde a dimensão de reconstrução:",
                                font=self.fontePadrao)
        self.dimensaoLabel.pack(side=LEFT)

        self.dimensao = Entry(self.terceiroContainer)
        self.dimensao["width"] = 30
        self.dimensao["font"] = self.fontePadrao
        self.dimensao.pack(side=LEFT)

        self.delayLabel = Label(self.quartoContainer,
                                text="Escolha o parâmetro tau, lembre-
se que ele corresponde ao delay da reconstrução:",
                                font=self.fontePadrao)
        self.delayLabel.pack(side=LEFT)

        self.delay = Entry(self.quartoContainer)
        self.delay["width"] = 31
        self.delay["font"] = self.fontePadrao
        self.delay.pack(side=LEFT)

        self.validar = Button(self.quintoContainer)
        self.validar["text"] = "Converter TS em SC"
        self.validar["font"] = ("Times", "10")
        self.validar["width"] = 20
        self.validar["command"] = self.converteTsinSC
        self.validar.pack()

    def converteTsinSC(self):
        serie = eval(self.tipoEntra.get())
        d = int(self.dimensao.get())
        tau = int(self.delay.get())

        plotSerie(serie)
        window_t = slidingWindows(d, tau, serie)
        if d == 4:
            plot4d(window_t)
        elif d == 3:
            plot3d(window_t)
        elif d == 2:
            plot2d(window_t)
        elif d == 1:
            print("Com esse valor de d não conseguiremos reconstruir o
espaço, que tal escolher um outro valor?!")
        else:
            plot4d(window_t)

        numpy_window = toNumpyArray(window_t)
        dist_matrix = getDistanceMatrix(numpy_window)
        simplicial_matrix = getEmptySimplicialMatrix(dist_matrix)

```



```

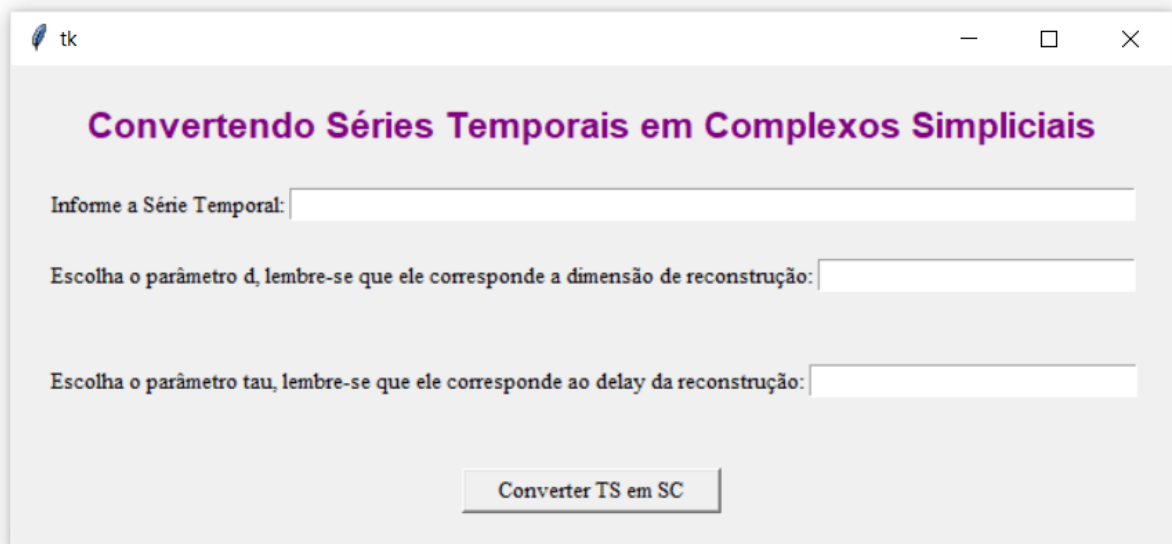
        simplicial_list = getSimplicialList(simplicial_matrix,
dist_matrix, 0.1, 0.8)
        unique = getUniqueSimplicialList(simplicial_list)
        list_of_tuples = getTuplesFromSimplicialList(unique)
        graph_list = getGraphList(list_of_tuples)
        print("Matriz de distâncias:", np.array_str(dist_matrix,
precision=3))
        plotGraph(graph_list[-1])

janela = Tk()
Interface(janela)
janela.mainloop()

```

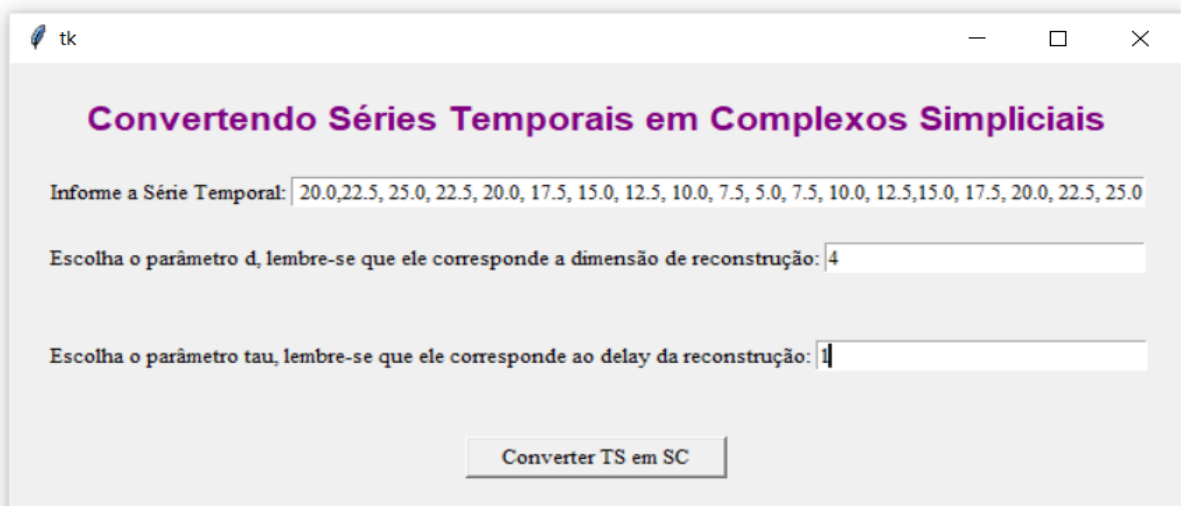
Interface do programa

O programa é iniciado



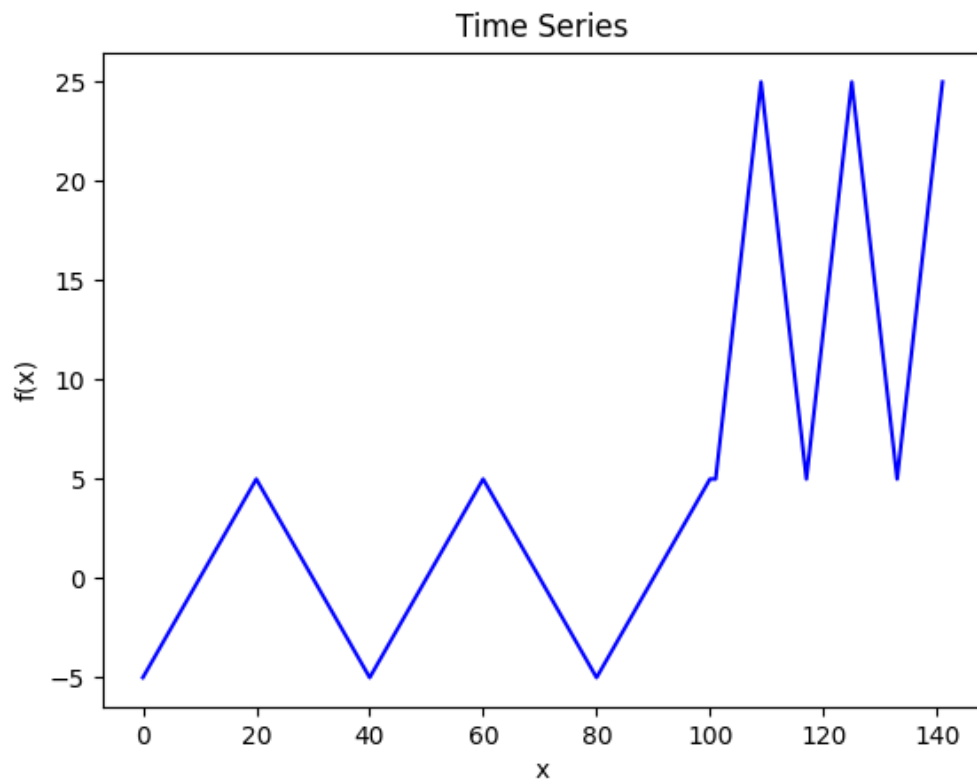
The screenshot shows a Tkinter window titled "tk" with standard window controls. The main title is "Convertendo Séries Temporais em Complexos Simpliciais" in purple. Below the title, there are three input fields with labels: "Informe a Série Temporal:", "Escolha o parâmetro d, lembre-se que ele corresponde a dimensão de reconstrução:", and "Escolha o parâmetro tau, lembre-se que ele corresponde ao delay da reconstrução:". At the bottom center, there is a button labeled "Converter TS em SC".

São colhidas as informações do usuário

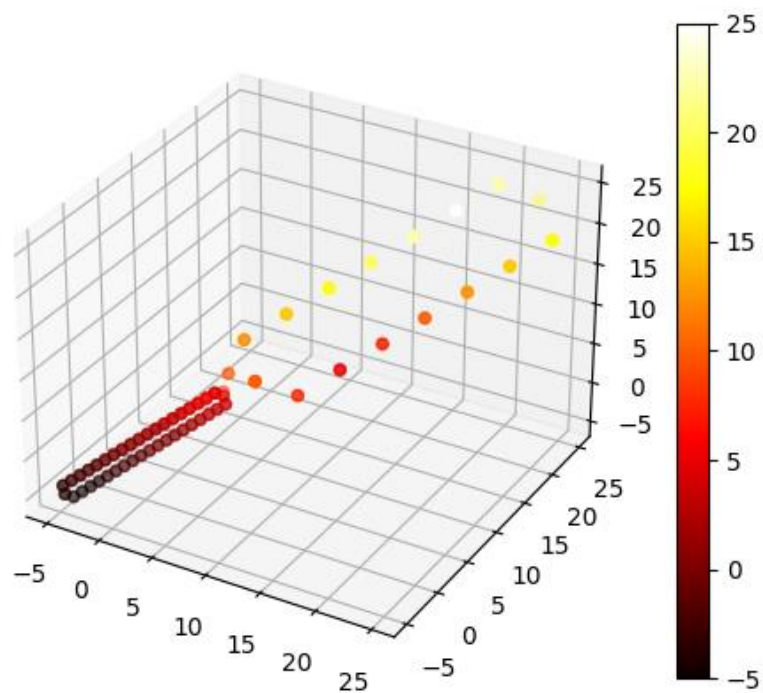


This screenshot shows the same interface as the previous one, but with user input. The "Informe a Série Temporal:" field now contains the text "20.0,22.5, 25.0, 22.5, 20.0, 17.5, 15.0, 12.5, 10.0, 7.5, 5.0, 7.5, 10.0, 12.5,15.0, 17.5, 20.0, 22.5, 25.0". The "Escolha o parâmetro d..." field contains the number "4". The "Escolha o parâmetro tau..." field contains the number "1". The "Converter TS em SC" button remains at the bottom.

Após o clique do botão são exibidos a série temporal



A série temporal reconstruída de acordo com os parâmetros fornecidos.



E o complexo simplicial completamente conectado.

