Universidade Federal de Pernambuco
Centro de Informática

Graduação em Engenharia da Computação

Random Networks of Carbon Nanotubes for Flexible Electronics: Modeling and Analysis

Jefferson Carlos Lima da Costa

Trabalho de Graduação

Recife
25 de setembro de 2023

Universidade Federal de Pernambuco
Centro de Informática

Jefferson Carlos Lima da Costa

Random Networks of Carbon Nanotubes for Flexible Electronics: Modeling and Analysis

Trabalho apresentado ao Programa de Graduação em Engenharia da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Engenharia da Computação.

Orientador: *Prof. Stefan Michael Blawid*

Recife
25 de setembro de 2023

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Costa , Jefferson Carlos Lima da .
Random networks of carbon nonotubes for flexible eletronics: modeling and analysis / Jefferson Carlos Lima da Costa . - Recife, 2023.
45 : il.

Orientador(a): Stefan Michael Blawid
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Engenharia da Computação - Bacharelado, 2023.
Inclui referências, apêndices.

1. Stick-percolation model. 2. Eletrônica flexível . 3. Nanotubo de carbono . 4. Redes aleatórias. 5. Resistores de filme fino. I. Blawid, Stefan Michael . (Orientação). II. Título.

620 CDD (22.ed.)

To my family and my friends, who stood by my side throughout this journey and never gave up on me, even when I gave up on myself—without you, none of this would have been possible.

Acknowledgements

First and foremost, I would like to thank my parents, José and Carla, along with my siblings, Priscila, Thamires, and Jessé. Their unwavering support has been my rock. My mother, in particular, has been my biggest believer, even when I had doubts about myself.

A special thanks to my Aunt Maria for her unwavering support. Her genuine interest in my academic pursuits means the world to me.

I would also like to thank the friends I made throughout my Computer Engineering course: Gabriel, Nathalia, Vitória, Kailane, Carlos, Daniel, Diogo, Thiago, Pedro, Mateus, and Lucas. You have been my pillars of strength during moments of uncertainty, and your camaraderie has made this journey enjoyable.

A heartfelt thank you to Romildo, who generously shared his time and expertise, and who helped me greatly during my learning of the Python programming language. Undoubtedly, these were moments that contributed significantly to my journey and led me to successfully develop this project.

My time at UFPE introduced me to more amazing friends: Elaine (whom I knew a year before entering UFPE but grew closer to during our college years), Endrew, Raissa, and Neemias. Our conversations, whether through WhatsApp or in person, were a source of comfort during challenging times. And let's not forget the countless lunches and dinners at RU; those memories will stay with me forever.

The UFPE also gave me another great friend, Marcela. We have had meaningful conversations on a wide range of topics, from football to college life and everything in between. Marcela, you may not realize it, but our talks have been a source of great support for me.

A warm thank you goes to my friend Alexaenny, whom I can affectionately call Xay. We have shared countless conversations, from academic matters to personal issues, and your support has been invaluable.

I would also like to express my gratitude to my friend Jennypher. Even though we have not met in person yet, I have a lot of trust in you and consider you a dear friend. Our conversations have been really meaningful to me and have provided significant support throughout this journey.

Last but certainly not least, I must acknowledge Professor Stefan. Words can hardly capture how vital your patience, mentorship, and guidance were throughout this process. Without you, this TCC would not have been possible. As I have mentioned before: you are the best, or as they say in German, "Du bist der beste" (I hope Google got that right).

Lastly, I want to extend my heartfelt gratitude to all the individuals who supported and guided me throughout this lengthy academic journey. Thank you all!

"É necessário sempre acreditar que o sonho é possível,
que o céu é o limite e você, truta, é imbatível.
Que o tempo ruim vai passar, é só uma fase,
e o sofrimento alimenta ainda mais a sua coragem.
Que a sua família precisa de você,
lado a lado para te apoiar se ganhar, se perder."
—RACIONAIS MC'S (A Vida É Desafio)

Resumo

A eletrônica flexível permite a criação de diversos fatores de forma, tais como formas macias, flexíveis, dobráveis, elásticas e leves. Esse avanço abre caminho para a ampla disponibilidade de inteligência ambiente. Aplicações promissoras foram demonstradas em vários campos, incluindo monitoramento de saúde, industrial, ambiental e estrutural. Para desenvolver com sucesso dispositivos eletrônicos flexíveis, principalmente sua interface com a unidade de computação e comunicação, é crucial desenvolver ferramentas apropriadas de automação de projeto (EDA) e projeto auxiliado por computador (TCAD). Essas ferramentas facilitarão a projeção de dispositivos eletrônicos flexíveis e garantirão uma boa integração. Um material promissor para a eletrônica flexível é uma Rede Aleatória (RN) de Nanotubos de Carbono (CNTs). O projeto visa desenvolver um modelo de stick-percolation para RNs e também calcular o transporte de elétrons através das RNs geradas pelo método de Monte-Carlo. Além disso, o projeto se concentrará no estudo das propriedades gerais de percolação das RNs.

Palavras-chave: Eletrônica flexível, Automação de Projeto Eletrônico, Redes Aleatórias, Nanotubos de carbono, Modelo de percolação, Método de Monte Carlo

Abstract

Flexible electronics enable the creation of various form factors, such as soft, flexible, foldable, elastic, and lightweight shapes. This advancement paves the way for the widespread availability of ambient intelligence. Promising applications have been demonstrated in several fields, including health monitoring, industrial, environmental, and structural domains. To successfully develop flexible electronic devices, especially their interface with the computing and communication unit, it is crucial to develop appropriate Electronic Design Automation (EDA) and Computer-Aided Design (CAD) tools. These tools will facilitate the projection of flexible electronic devices and ensure seamless integration. A promising material for flexible electronics is a Random Network (RN) of carbon nanotubes (CNTs). The project aims to develop a stick-percolation model for RNs and also to calculate electron transport through RNs generated by the Monte Carlo method. Furthermore, the project will focus on studying the general percolation properties of RNs.

Keywords: Flexible electronics, Electronic Design Automation, Random Networks, Carbon nanotubes, Stick-percolation model, Monte Carlo method

Contents

List of Figures

CHAPTER 1

Introduction

Flexible electronics have the potential to revolutionize the way we interact with technology. Traditional rigid electronics are often bulky and cumbersome, and they can be difficult to use in certain environments. On the other hand, flexible electronics are devices capable of assuming soft, flexible, foldable, elastic, and lightweight shapes while maintaining optimal performance and reliability [1]. This progress sets the stage for the widespread availability of ambient intelligence.

Notably, promising applications have emerged across various domains. For instance, in health monitoring, a notable development is a patch sensor designed to detect blood glucose concentration [2]. In the industrial sector, these electronics find purpose in wearables housing health and habit-monitoring sensors, as well as in implantable electronics that advance medical imaging and diagnostics [1]. Additionally, they exhibit substantial benefits in environmental contexts, yielding enhanced resource efficiency, waste reduction, and sustainability outcomes [3].

Furthermore, within structural domains, embedding sensors within building materials facilitates real-time tracking of a structure's health. These sensors excel in identifying shifts in temperature, humidity, and other environmental factors capable of impacting a building's integrity [4]. Moreover, they are adept at promptly detecting cracks and other signs of damage, thus enabling early identification and timely repairs [4].

People involved in the creation and design of electronic circuits need software that allows them to first create and simulate these circuits using computers. In order to make progress in developing flexible electronic devices, we rely on two important types of tools: Electronic Design Automation (EDA) [5] and Computer-Aided Design (CAD) [6]. These software tools are provided by the industry, such as Synopsys, but we can also do this in college. If a designer, for instance, requires a circuit with a resistance of 1 kΩ, and wishes to ascertain the appropriate process parameters needed to achieve this resistance, like the density (number of sticks) and the size of sticks for random network based materials, this kind of software should be capable of providing such information. These tools will facilitate the design of flexible electronic devices and ensure seamless integration [7].

A promising material for flexible electronics is a Random Network (RN) of carbon nanotubes (CNTs) [8]. The studies referenced by [9] discuss how, from the time of their discovery in 1991, carbon nanotubes (CNTs) have consistently attracted significant attention as a promising electronic material, primarily due to their exceptional electrical properties.

This project focuses on the construction and analysis of Random-Line-Graphs (RLGs) as a model for RN based materials using Python. RLGs are generated by assigning geometric coordinates to individual lines, resembling random-geometric graphs.

The process involves random point generation, angle selection, and line length assignment within a unit square. Intersections between lines are identified by subdividing regions in the unit square and implementing functions for region-based checks.

Path-finding is carried out using a breadth-first search algorithm, filtering nodes of interest. Impedance calculation involves matrix operations and voltage-current relationships. However, challenges arise due to singular Laplacian matrices caused by "dead-end points" in the percolation paths, requiring iterative pruning for resolution.

This project offers insights into RLG modeling and impedance calculations with potential applications in diverse fields.

The experiments conducted involve varying the density, represented by the number of lines (N), and the length of line segments (a). The objective is to determine the critical values of N and a at which percolation paths emerge.

Initial experiments with a fixed a of 0.06 and varying N revealed that even when N reached 800, with a significant number of intersections, no percolation path emerged. However, for $N = 2000$, with over double the number of intersections compared to lines, a percolation path was identified, reaffirming the relationship between the number of lines and intersections.

Subsequently, one of the experiments with a fixed N of 800 and varying a demonstrated that longer line segments, such as $a = 0.12$, significantly increased intersection occurrences, leading to the discovery of a percolation path. These findings underscore the influence of both N and a on percolation phenomena in the Random Network of Carbon Nanotubes.

1.1 Objectives

The aim of this project is to develop a computer-aided design tool for thin-film resistors composed of a material that forms a random network of short conducting connections, e.g., polymers, nanowires or -tubes. To this extent, this project seeks to generate Random-Line-Graphs (RLGs) by the Monte Carlo Method [8], to analyze the RLG connectivity, and to solve the graph Laplacian with weighted edges resulting in a stick-percolation model for thin-film resistors.

The specific objectives of this project are:

- Identify the value of N and the value of a for which a percolation pathway starts to exist.
- Compute the impedance of the pathways.

1.2 Outline

The upcoming chapters are structured as follows:

- **Chapter 2**: Introduction of essential concepts for understanding the purpose of this work.
- **Chapter 3**: Description of the development of the proposal, presentation of the tools used, detailed explanation of the model's implementation.
- **Capítulo 4**: Experiments and results.

- **Capítulo 5**: Conclusion and next steps
- **Capítulo 6**: Appendix

CHAPTER 2

Theoretical foundation

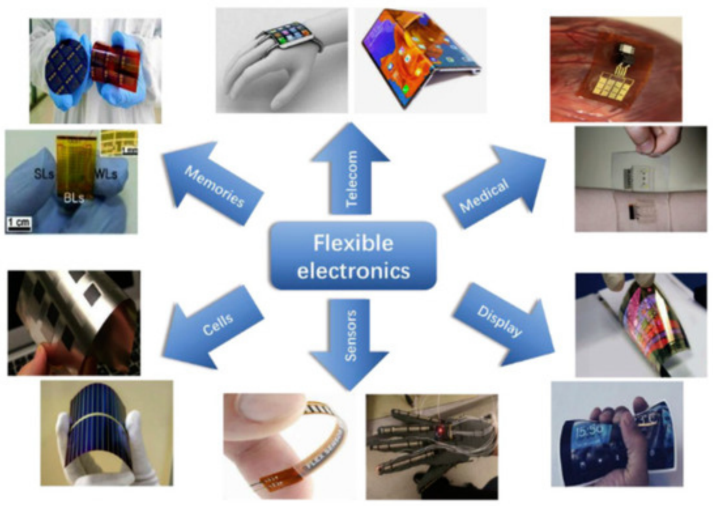
2.1 What is flexible electronic?

Flexible electronics encompass electronic components and circuits deliberately designed to possess a lightweight, thin profile while retaining their operational capabilities even when subjected to bending or stretching. These innovations are fabricated on pliable materials like plastics and polymers, granting them the ability to conform to a range of shapes and find utility in various applications [1].

Flexible electronics serve a wide range of purposes, including flexible displays, wearable technology, sensors, solar panels, and much more [1], as the examples of application that can be seen in Figure 2.1. Although they offer advantages like portability and durability, they come with their share of challenges, such as intricate manufacturing and performance limitations when compared to rigid electronics [1].

Nevertheless, ongoing research and development are paving the way for exciting innovations in this field [1], creating new avenues for seamlessly integrating electronics into our everyday lives.

Figure 2.1 A few applications of flexible electronics in everyday life.



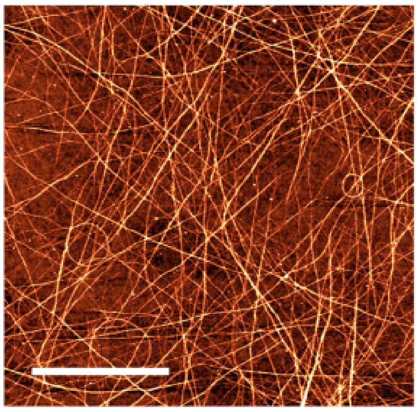
Source: SALEH, et al., 2021, p. 2.

2.2 What is a Random Network (RN) of carbon nanotubes (CNTs)?

A Random Network (RN) of carbon nanotubes, an example of which can be seen in Figure 2.2, is a type of electronic material in which carbon nanotubes are interconnected randomly with each other. These networks can be formed by single-wall carbon nanotubes (SWNTs) or multi-walled carbon nanotubes (MWCNTs) [10], which consist of a nesting of SWNTs [11].

Carbon nanotubes (CNTs) can be classified into two types: metallic and semiconducting, and they are widely recognized for their good electrical properties [9]. These electrical properties depend on factors such as the uniformity of the carbon nanotube diameter, their purity, and density [12].

Figure 2.2 A random network of SWNTs on SiO2. The scale bar is 1 μm.



Source: ZORN, et al., 2021, p. 3.

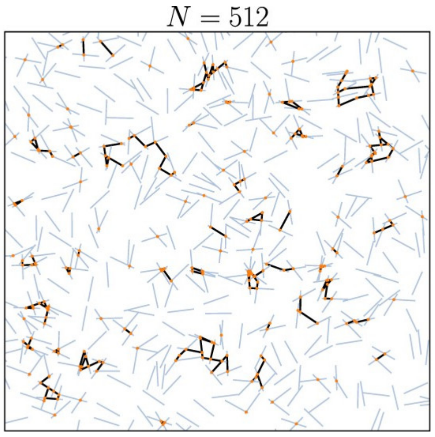
2.3 What is a Random-Line-Graph (RLG)?

A Random-Line-Graph (RLG) is a graph model that emerges from a possible interaction of line segments in a spatial network. In this work, we call these line segments also "sticks". In this model, intersections are the points at which two line segments intersect, creating nodes. Nodes are linked by edges if they share a common line segment. As the value of N (number of lines) increases, the probability of intersections occurring also increases, leading to a more interconnected Random Line Graph (RLG) with fewer isolated nodes [13].

In this project, RLGs will be used to represent Random Carbon Nanotube Networks, where each line of the RLG represents a carbon nanotube.

In the example in Figure 2.3, we have an RLG with 512 lines. The black lines (edges) are the ones that contain an intersection with some other line, while the orange points (nodes) are the intersection points between the lines.

Figure 2.3 RLG that results from the intersection of 512 line segments (sticks)



Source: BÖTTCHER, 2020, p. 2.

2.4 What is percolation in nanotube networks?

Percolation in nanotube networks is a fascinating occurrence. It happens when the overall electrical conductivity of a structure made from carbon nanotubes (CNTs) gets a significant boost as the density of these nanotubes in the network reaches a specific critical point [14]. Below this threshold, the network's electrical conductivity is pretty much nonexistent. This phenomenon is quite similar to percolation theory [15], which is often used to explain how fluids flow through porous materials. In both cases, there's a critical threshold where a continuous pathway for fluid flow or electrical conductivity emerges [14].

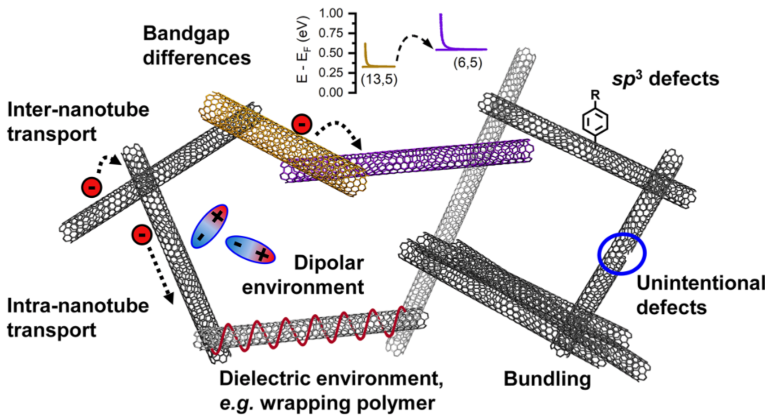
When we talk about carbon nanotubes, percolation happens when a bunch of these tiny tubes connect in such a way that it forms a pathway for electrons to flow from one end to the other [16]. As you add more nanotubes to this network, you get more and more pathways for the electrons to move through, and that's when you see a big jump in how well an electrical current can flow [16].

2.5 How does current flow in carbon nanotube networks?

Charge carriers, which are essentially the particles responsible for transporting electrical charge, navigate through carbon nanotubes in a variety of ways. Their movement involves passing through the connections between nanotubes, a process often referred to as inter-nanotube transport, as well as traveling along the individual segments of a nanotube, known as intra-nanotube transport [14].

Several factors come into play and influence this intricate process. These factors include differences in bandgaps between various nanotubes, the presence of a dipolar environment – for example, at the interface between a semiconductor and a dielectric material, the effects of the dielectric environment resulting from leftover wrapping polymers used in the nanotube selection process, and the scattering of charge carriers due to both unintentional defects that may occur during growth or processing and purposeful sp3 defects. All of these factors together contribute to the complex behavior exhibited by charge carriers as they move within carbon nanotubes [14]. Figure 2.4 provides a visual representation of the different elements that contribute to either enhancing or impeding conductance within a densely interconnected nanotube network.

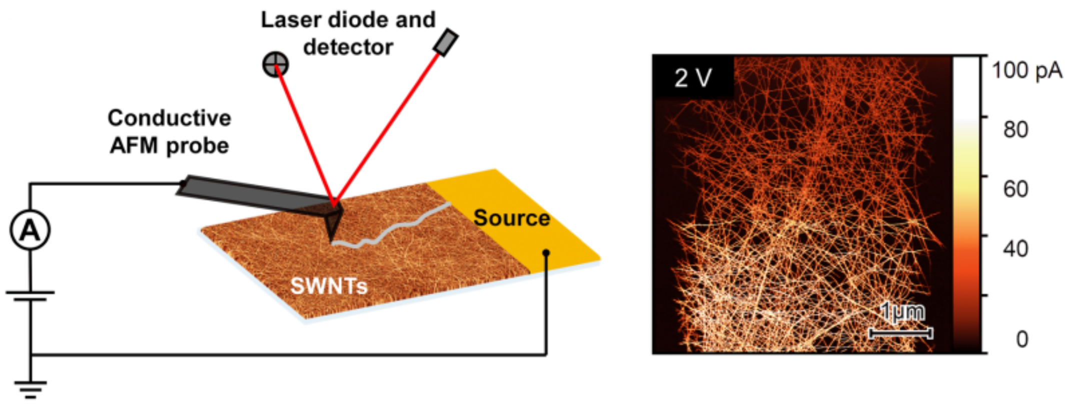
Figure 2.4 Factors affecting conductance in semiconducting SWNT Networks



Source: ZORN, et al., 2021, p. 7.

The current in a carbon nanotube RNN will flow in complex patterns. At the intersection point the current will branch, exploring different parts of the network, and get united again. There exists an ingenious technique to make this flow visible, see Figure 2.5. By contacting single carbon nanotubes from the whole film, the whole connected path will "light up". These currents flow only in a part of the RNN and their path is very similar to the connected paths that will be constructed and studied in the present work.

Figure 2.5 An ingenious technique to make this flow visible



Source: ZORN, et al., 2021, p. 7.

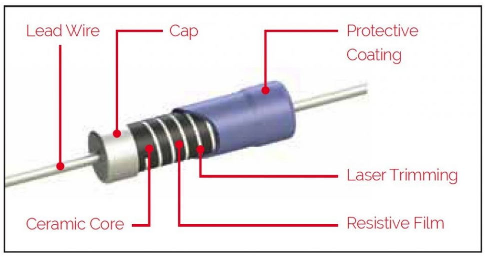
2.6 What is a stick-percolation model?

The stick-percolation model is a simplified theoretical approach employed to describe the characteristics of nanotube networks within the context of percolation. In this model, nanotubes are abstracted as idealized "sticks" that establish random connections or junctions in space. It posits that electron transportation predominantly occurs along these sticks, and the percolation threshold is attained when a sufficient density of sticks forms a path [14].

2.7 What is a thin-film resistor?

A thin-film resistor, as exemplified in Figure 2.6, is an electronic component employed to regulate the flow of electric current. It is composed of an extremely thin layer of resistive material applied to a substrate. These resistors are renowned for their high precision, stability, and resilience to temperature fluctuations. They find extensive use in precision electronic circuits due to their accuracy and compact size. [17].

Figure 2.6 Construction of a thin film resistor



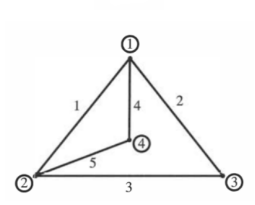
Source: HOVSEPIAN, 2021, p. 1.

2.8 What is the oriented incidence matrix of an undirected graph?

The oriented incidence matrix of an undirected graph is like a map that shows how the graph's connections work as if they had a direction. This matrix is organized in a way that it has |V| columns and |E| rows, where |V| represents the number of vertices (points), and |E| is the number of edges (lines). In Figure 2.7, an example of this type of matrix can be seen.

In this matrix, the numbers can be either -1, 0, or 1. Specifically, for each vertex, if it is where an edge starts (the "tail"), we put a -1 in the corresponding spot. If it is where an edge ends (the "head"), we use a 1. And if the vertex is not connected to that edge, we mark it with a 0.

Figure 2.7 An example of an oriented incidence matrix of an undirected graph.



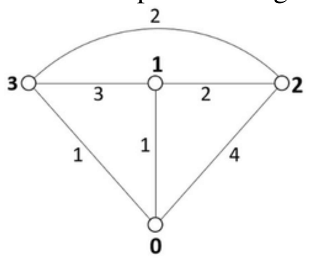
$$A_{\text{Tri}} = \begin{bmatrix} -1 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ -1 & 0 & 0 & 1 \\ 0 & -1 & 0 & 1 \end{bmatrix}$$

Source: BLAWID, 2023.

2.9 What is a Laplacian graph with weight edges?

A graph Laplacian with weighted edges is a way to mathematically represent a graph by attaching numerical weights to each of the edges connecting its nodes. A matrix that can be constructed from this type of graph is the Laplacian matrix, which reflects these weighted relationships: the diagonal entries denote the sum of the weights linked to each node, while the off-diagonal entries represent the connections between adjacent nodes. An example of a weighted graph Laplacian and a Laplacian matrix can be seen in Figure 2.8.

Figure 2.8 An example of a weighted graph Laplacian and a Laplacian matrix.



$$Y = \begin{array}{c|cccc} & 1 & 2 & 3 & 0 \\ \mathbf{1} & 6 & -2 & -3 & -1 \\ \mathbf{2} & -2 & 8 & -2 & -4 \\ \mathbf{3} & -3 & -2 & 6 & -1 \\ \mathbf{0} & -1 & -4 & -1 & 6 \end{array}$$

Source: SHARMA, et al., 2019, p. 4.

2.10 What is a Computer-Aided Design (CAD) tool?

A Computer-Aided Design (CAD) application stands as a software tool that empowers engineers, architects, and designers to construct and amend digital representations of physical objects or architectural designs [18][19].

In comparison to traditional manual drafting methods, CAD software offers a host of benefits, including heightened precision, elevated accuracy, increased efficiency, better communication, the ability to simulate real-world scenarios, and support for manufacturing processes [18][19].

Across a multitude of industries, CAD tools occupy a pivotal role in the toolkit of professionals, enabling the generation of diverse model types such as 2D drawings, 3D models, assembly representations, engineering schematics, manufacturing plans, and technical illustrations [18][19][20].

2.11 What is the Monte Carlo Method?

The Monte Carlo method is a computer-based technique used to estimate solutions for complex mathematical or statistical problems. It works by generating random inputs, running simulations, and then combining the results to provide approximations. It finds applications in different fields, including finance, economics, physics, engineering, physical processes and structures, statistics, random graphs, and combinatorial structures [21]. In this project, it is used to generate a Random-Line-Graph.

CHAPTER 3

Methods

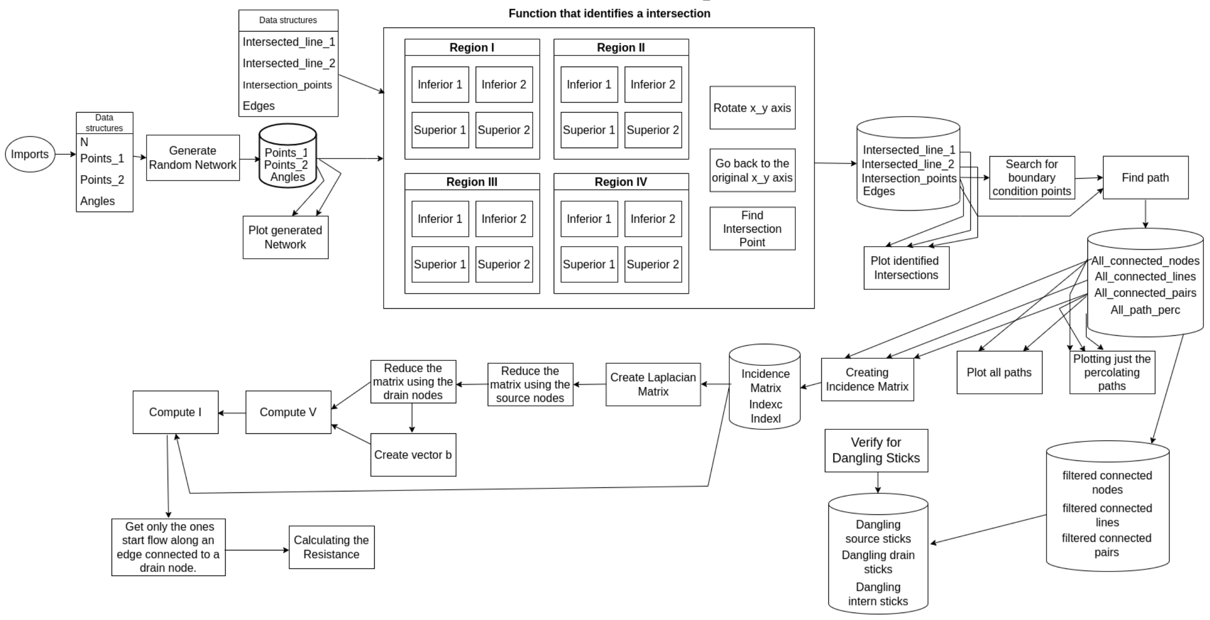
3.1 How is the stick-percolation model implemented?

In the implementation of the model developed in this project, as can be seen in the flowchart in Figure 3.1, the main data structure used was the list. Lists were used, for example, to store the coordinates of the points that make up the random networks, to store the nodes that make up the path, and also to store the lines connected to these nodes.

On the other hand, matrices were used in the creation of the incidence matrix and the Laplacian matrix. In addition, numpy arrays were used to store, for example, the voltages of the internal nodes and the computed currents.

Several functions were implemented, such as the one that generates the random network, the functions for plotting the intersections, and the one that plots the found paths. Functions were also implemented that make up the module responsible for identifying intersections, as well as those used to determine if a line segment is within a region subdivision and the function that performs the rotation of the coordinate system.

Figure 3.1 Flowchart of the stick-percolation model



Function that identifies a intersection
Data structures
Intersected_line_1
Intersected_line_2
Intersection_points
Edges
Imports
Data structures
N
Points_1
Points_2
Angles
Generate Random Network
Points_1 Points_2 Angles
Plot generated Network
Region I
Region II
Region III
Region IV
Inferior 1
Inferior 2
Superior 1
Superior 2
Rotate x_y axis
Go back to the original x_y axis
Find Intersection Point
Intersected_line_1 Intersected_line_2 Intersection_points Edges
Search for boundary condition points
Find path
Plot identified Intersections
All_connected_nodes All_connected_lines All_connected_pairs All_path_perc
Creating Incidence Matrix
Plot all paths
Plotting just the percolating paths
Incidence Matrix Indexc IndexI
Create Laplacian Matrix
Reduce the matrix using the source nodes
Reduce the matrix using the drain nodes
Create vector b
Compute V
Compute I
Get only the ones start flow along an edge connected to a drain node.
Calculating the Resistance
Verify for Dangling Sticks
Dangling source sticks Dangling drain sticks Dangling intern sticks
filtered connected nodes filtered connected lines filtered connected pairs

Source: Author, 2023.

3.2 How is a Random-Line-Graph constructed?

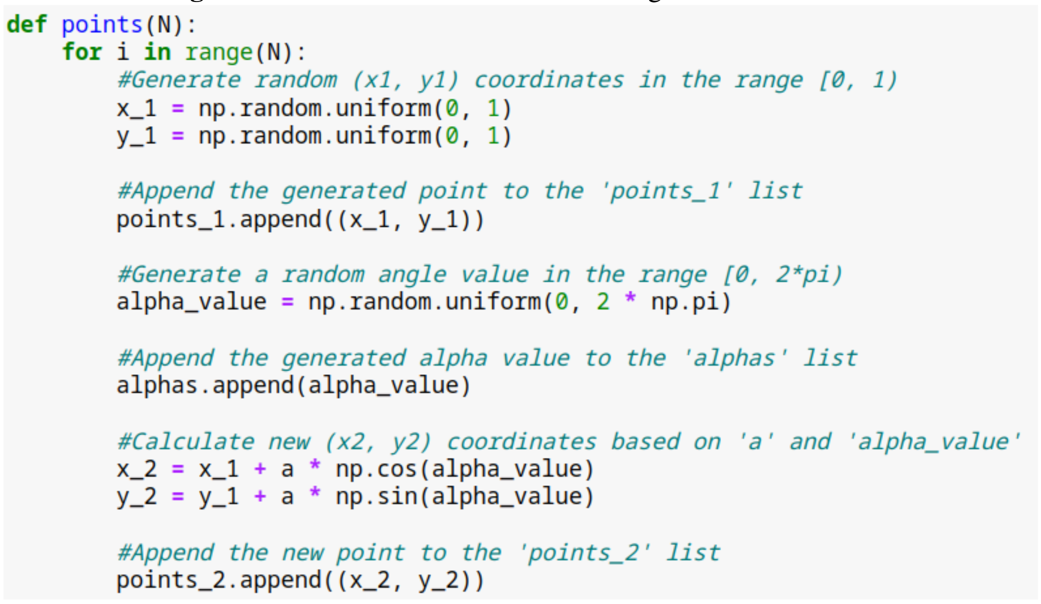
RLGs are constructed by assigning random geometric coordinates to each of the lines that compose it, very similar to the construction of a random-geometric graph (RGG) [13][22]. In this study, we generated the coordinates for the lines comprising the graph by following the methodology outlined in the article "A Random-Line-Graph Approach to Overlapping Line Segments" (Bottcher, 2020) [13].

For the starting point (x_1, y_1) of each line, we also used a uniform distribution, meaning $x_1, y_1 \sim \mathrm{U}(0,1)$, and the corresponding endpoint is generated from $(x_2, y_2) = (x_1 + a * \cos(\theta), y_1 + a * \sin(\theta))$, where $\theta \sim \mathrm{U}(0, 2\pi)$.

In the development of this project, the Python programming language and its libraries were used. Some of the libraries used were Random, Numpy, Math, Matplotlib, and Collections.

We began by implementing the function that will generate the RLGs. In Figure 3.2, it is possible to see the implementation of this function. The function works as follows: given an integer N, it generates two sets of random points, one containing the starting points and the other containing the endpoints that form each line. Each iteration in a loop executed N times generates a tuple of random points (x_1, y_1) representing the starting point and stores it in an array. Then, a randomly selected angle, with a value between 0 and 2π, is calculated. A new point (x_2, y_2) is calculated by moving a distance a (length of the line (stick)) in the direction specified by the chosen angle, and this point is stored in another array. The values of the chosen angles are saved into another array.

Figure 3.2 The code of the function that generates the RLGs:



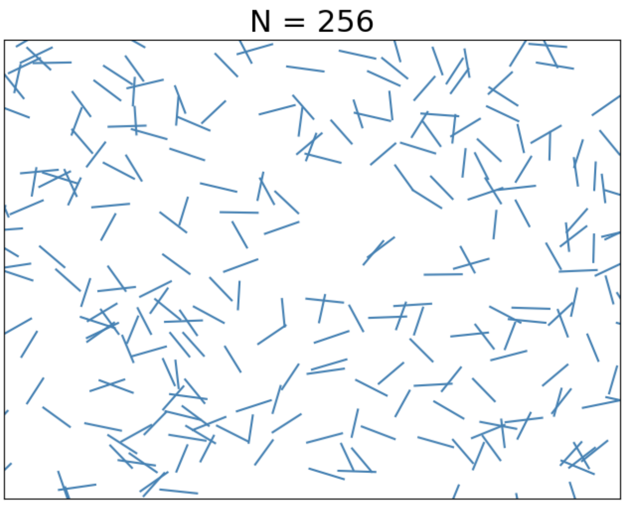
```python
def points(N):
    for i in range(N):
        #Generate random (x1, y1) coordinates in the range [0, 1)
        x_1 = np.random.uniform(0, 1)
        y_1 = np.random.uniform(0, 1)

        #Append the generated point to the 'points_1' list
        points_1.append((x_1, y_1))

        #Generate a random angle value in the range [0, 2*pi)
        alpha_value = np.random.uniform(0, 2 * np.pi)

        #Append the generated alpha value to the 'alphas' list
        alphas.append(alpha_value)

        #Calculate new (x2, y2) coordinates based on 'a' and 'alpha_value'
        x_2 = x_1 + a * np.cos(alpha_value)
        y_2 = y_1 + a * np.sin(alpha_value)

        #Append the new point to the 'points_2' list
        points_2.append((x_2, y_2))
```

Source: Author, 2023.

It was necessary to implement a function to plot the generated lines and thus obtain our random network of carbon nanotubes. This function generates a graph using the 'Matplotlib' library. It connects pairs of points from the two arrays that contain the points, forming the lines that make up our network.

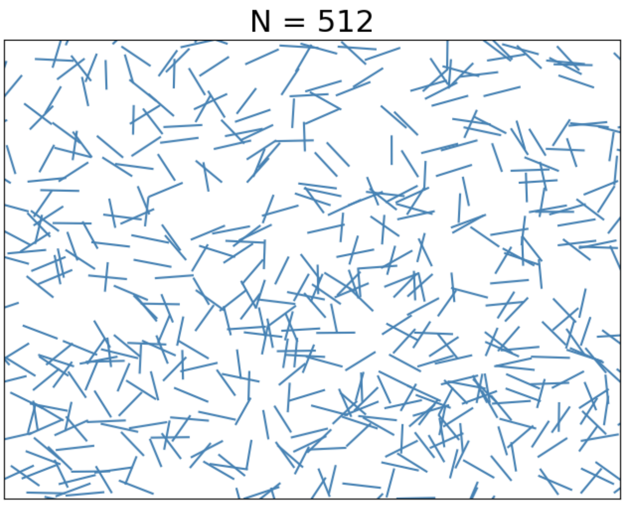
Below, we have some examples of random networks generated by the function. In Figure 3.3 it is a random network for 256 lines (sticks), and in Figure 3.4 lines (sticks), and in Figure 3.5 1024 lines (sticks).

Figure 3.3 RLG generated by the implemented function in this project for 256 lines (sticks)



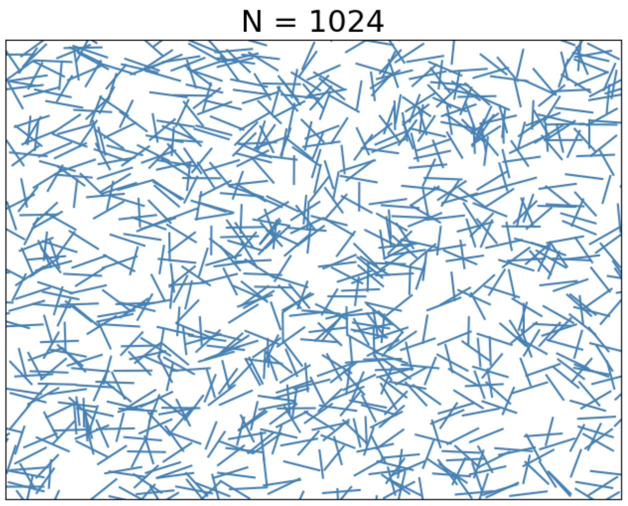
Source: Author, 2023.

Figure 3.4 RLG generated by the implemented function in this project for 512 lines (sticks)



Source: Author, 2023.

Figure 3.5 RLG generated by the implemented function in this project for 1024 lines (sticks)



Source: Author, 2023.

3.3 How are intersections between lines (sticks) being identified?

We know that two line segments (sticks) intersect if they are both in the same region [13]. The definition of each region, i.e., the values of x, y, and θ (see (Appendix) for more details), corresponds to area segments in the unit square. Each region has been subdivided into four parts, corresponding to the lower left, upper left, lower right, and upper right sides.

For each of these subdivisions, we have implemented a Python function that allows us to check whether a given line, defined by its length a, coordinates x and y, and angle θ, is or is not in that region. In figure 3.6, it is possible to see the implementation of one of these functions, more specifically the one that checks the subdivision we refer to as 'the left-lens-shaped corner of the lower square-shaped region'.

A simplification was made: another function was created in which the four functions related to subdivisions are called, making it possible to use a single function to check whether a line segment is or is not in a region. In Figure 3.7, it is possible to see how this simplification was made.

Figure 3.6 The code of the function that checks a subdivision:

```python
def is_point_inside_region_1_inferior_1(x, y, theta, a):
    # Check if the point (x, y) is within the specified region
    if x >= 0 and x <= a/2:
        # Calculate the lower limit of y based on the equation of a circle
        lower_limit_y = a - math.sqrt(a**2 - (x - a)**2)

        # Check if y is within the defined range
        if y >= lower_limit_y and y <= a:
            # Calculate two angles, atan_1 and atan_2
            atan_1 = math.atan((a - y) / (a - x))
            atan_2 = math.pi/2 + math.atan(x / (a - y))

            # Check if the given theta is within the angle range
            # defined by atan_1 and atan_2
            if theta >= atan_1 and theta <= atan_2:
                # If all conditions are met, return True,
                # indicating that the point is inside the region
                return True

    # If any condition is not met, return False,
    # indicating that the point is outside the region
    return False
```

Source: Author, 2023.

Figure 3.7 The code of the function that checks the lens-shaped region:

```python
def is_point_inside_region_1(x, y, theta, a):
    # Check if the point (x, y) is inside any of the four subregions of region 1
    region_1_inferior_1 = is_point_inside_region_1_inferior_1(x, y, theta, a)
    region_1_inferior_2 = is_point_inside_region_1_inferior_2(x, y, theta, a)
    region_1_superior_1 = is_point_inside_region_1_superior_1(x, y, theta, a)
    region_1_superior_2 = is_point_inside_region_1_superior_2(x, y, theta, a)

    # Return True if the point is inside any of the four subregions, otherwise, return False
    return region_1_inferior_1 or region_1_inferior_2 or region_1_superior_1 or region_1_superior_2
```

Source: Author, 2023.

The identification of the intersection between two lines is done by taking a line segment (stick), which for didactic purposes will be called line 1, as a reference and placing it horizontally at point (0,0) in the coordinate system. The second step is to check if any of the other lines, which are also, for didactic purposes, each referred to as line 2 when being used to verify whether they intersect with line 1 or not.

A Python function was implemented to perform this verification. In this function, a line is taken as a reference, and a loop is used to check if any of the other lines intersect it. Since a rotation is performed for this check to be done correctly, it is necessary to rotate the coordinate system so that we have the correct coordinates, i.e, the correct values of (x, y), to pass to functions such as the one in Figure 3.6 that checks if the line segment is or is not in a certain region.

If the function returns "true", that means there is an intersection, so the coordinates of the two lines that are crossing each other are saved in arrays. Additionally, the point at which these lines intersect is found using a function that was implemented based on equations (1) and (2). The implementation of this function that finds the point can be seen in Figure 3.8.

To be able to plot this intersection point, it is necessary to return to the original coordinate system. Both the rotation and the return to the coordinate system are done through functions that were implemented and are part of our stick-percolation model.

The intersection point found now in the original coordinate system is saved in an array that contains only the intersection points. The edges connected to the found intersection point are also saved. We will need these edges when we perform a search for paths in the graph.

$$x = x_{20} + \frac{a - y_0}{\tan(\phi_{20})} \tag{1}$$

$$y = y_{20} + \tan(\phi_{20}) \cdot (x - x_{20}) \tag{2}$$

Figure 3.8 The code of the function that finds the intersection points:

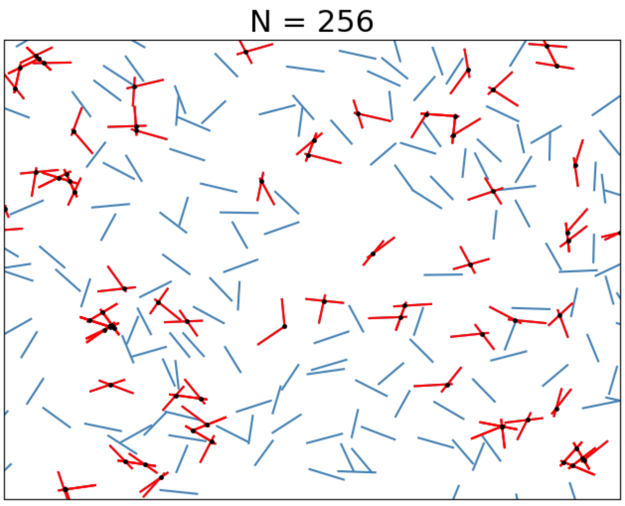
```python
def find_intersection_point(x2, y2, a, angle):
    try:
        # Calculate the x-coordinate of the intersection using trigonometry
        x = x2 + ((a - y2) / math.tan(angle))

        # Calculate the y-coordinate of the intersection using trigonometry
        y = y2 + math.tan(angle)*(x - x2)

        # Return the coordinates of the intersection point
        return x, y
    except ZeroDivisionError:
        # Handle the case where the tangent of the angle is zero,
        # which would cause a division by zero error
        print("Error: Tan(phi) cannot be zero.")

        # Return None to indicate an error condition
        return None
```

Source: Author, 2023.

Below, we provide a few illustrative instances of intersections identified through the function that was implemented as an integral part of the model developed within this project. The red lines, which correspond to edges, visually denote the locations where intersections occur, while the presence of a black dot, representing nodes, signifies the specific points at which these intersections take place.

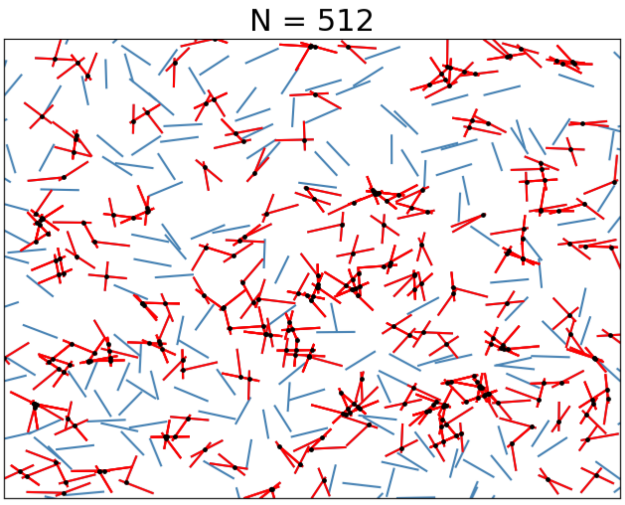
In Figure 3.9, the intersections identified for 256 lines (sticks) are shown, while in Figure 3.10, the intersections for lines (sticks) are displayed, and in Figure 3.11, intersections for 1024 lines (sticks) are presented.

Figure 3.9 Intersection identified by the implemented function in this project for 256 lines (sticks)



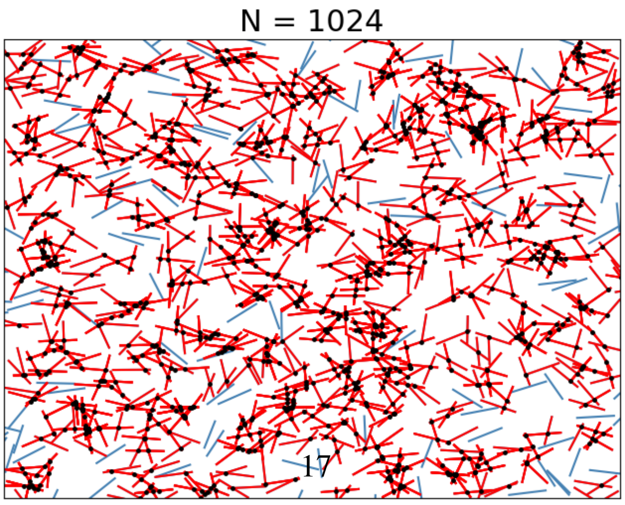
Source: Author, 2023.

Figure 3.10 Intersection identified by the implemented function in this project for 512 lines (sticks)



Source: Author, 2023.

Figure 3.11 Intersection identified by the implemented function in this project for 1024 lines (sticks)



Source: Author, 2023.

3.4 How to find a path in the generated graphs?

In this project, some of the randomly generated RNs can form a graph with a substantial number of nodes. Consequently, we need to filter out specific nodes, namely those in the source and drain regions.

The source nodes are defined as those with an x-value less than 0.1, while the drain nodes have an x-value greater than 0.9. Our objective is to identify paths in the graphs, particularly those starting from the left end and extending to the right end.

To accomplish this, we implemented a Python function that employs a breadth-first search (BFS) algorithm. The BFS algorithm is designed to uncover a path in the graph, starting from a given initial node, which, in this project, corresponds to a source node. It maintains sets and lists to keep track of visited nodes and records information pertaining to the path.

During its operation, the function iteratively explores the graph, marking visited nodes and identifying the edges along the path. Additionally, it checks a specific condition related to intersection points to ascertain the completeness of the path.

The function provides four key pieces of information as output: the visited nodes in order, the visited edges, pairs of nodes representing the visited edges, and a flag indicating whether the path found is a percolation path or not. A percolation path is a path that starts at the left end and extends to the right end of the graph.

3.5 How is the impedance being calculated?

Impedance calculation can be performed using the model developed in this project. Some pieces of the code implemented will be shown as the step-by-step calculation is explained, and a link to a GitHub repository containing the project can be found in the (Appendix).

Once we have the percolating paths, we can begin the calculation. The first step is to find the representation of the incidence matrix for each of the paths in the percolating graph, which, for didactic purposes, will be referred to as matrix A.

The nodes of the graph correspond to the columns of the matrix, while the edges correspond to the rows. In this matrix A, the numbers can be -1, 0, or 1. Specifically, for each vertex, if it is where an edge starts (the 'tail'), we place a -1 in the corresponding position. If it is where an edge ends (the 'head'), we use a 1. And if the vertex is not connected to that edge, we mark it with a 0.

The implemented function to create this matrix can be seen in Figure 3.12.

Figure 3.12 Function that creates matrix A

```python
def matrixA():
    # This function constructs a matrix 'ad' and two index arrays 'indexc' and 'indexl'.

    for k, perc in enumerate(all_path_perc):
        if perc == 1:
            # If 'perc' is equal to 1, it means there is a valid connection.

            # Extract connected nodes, pairs, and lines for this connection
            connected_nodes = all_connected_nodes[k]
            connected_pairs = all_connected_pairs[k]
            connected_lines = all_connected_lines[k]

            # Initialize variables to track unique connected lines
            vll = []  # List to store unique connected lines
            indexc = connected_nodes  # Index array for connected nodes
            numbc = len(connected_nodes)  # Number of connected nodes
            numbl = 0  # Initialize the number of connected lines

            # Iterate through connected nodes
            for i in range(0, numbc):
                # Iterate through lines associated with each node
                for ll in connected_lines[i]:
                    if ll not in vll:
                        numbl += 1
                        vll.append(ll)

            indexl = vll  # Index array for connected lines

            # Create an empty matrix 'ad' to represent the adjacency relationship
            ad = np.zeros((numbl, numbc))

            mc = -1  # Initialize a counter for connected nodes
            visited_lines = []  # List to keep track of visited lines
            for i in range(0, numbc):
                mc += 1
                # Iterate through lines associated with each node
                for ll in connected_lines[i]:
                    if ll in visited_lines:
                        ml = visited_lines.index(ll)
                        ad[ml][mc] = 1  # Set the adjacency matrix value to 1 if the line is visited
                    else:
                        visited_lines.append(ll)
                        ml = visited_lines.index(ll)
                        ad[ml][mc] = -1  # Set the adjacency matrix value to -1 if the line is newly visited

    return ad, indexc, indexl

# Note: This function appears to construct an adjacency matrix 'ad' based on information about connected nodes,
# pairs,
# and lines. It also creates index arrays 'indexc' and 'indexl' to reference connected nodes and lines.
```

Source: Author, 2023.

Once the incidence matrix corresponding to the given graph is presented, it is necessary to multiply it by its transpose. This multiplication yields the Laplacian matrix, which, for didactic purposes, will be referred to as matrix L. In this matrix L, unlike matrix A, the nodes correspond to both the rows and the columns.

The matrix A, which is our incidence matrix, contains node voltages. We need this matrix because it is essential to determine the voltage drop across each edge. However, to find the impedance of the paths, we need to apply Kirchhoff's law to the nodes. In other words, the algebraic sum of all currents entering or leaving a node in an electrical circuit must equal zero. We can state it as follows: let $I = C * A$, then $A^T * I = 0$, where C is the conductance matrix.

The matrix L is used to represent Kirchhoff's law. In this project, we assume that conductance is a constant value, i.e, $C = k$ (constant). Therefore, we can express Kirchhoff's law for the currents as follows: since $C = k$, then $A^T * (C * A) = k * A^T * A = k * L = 0$.

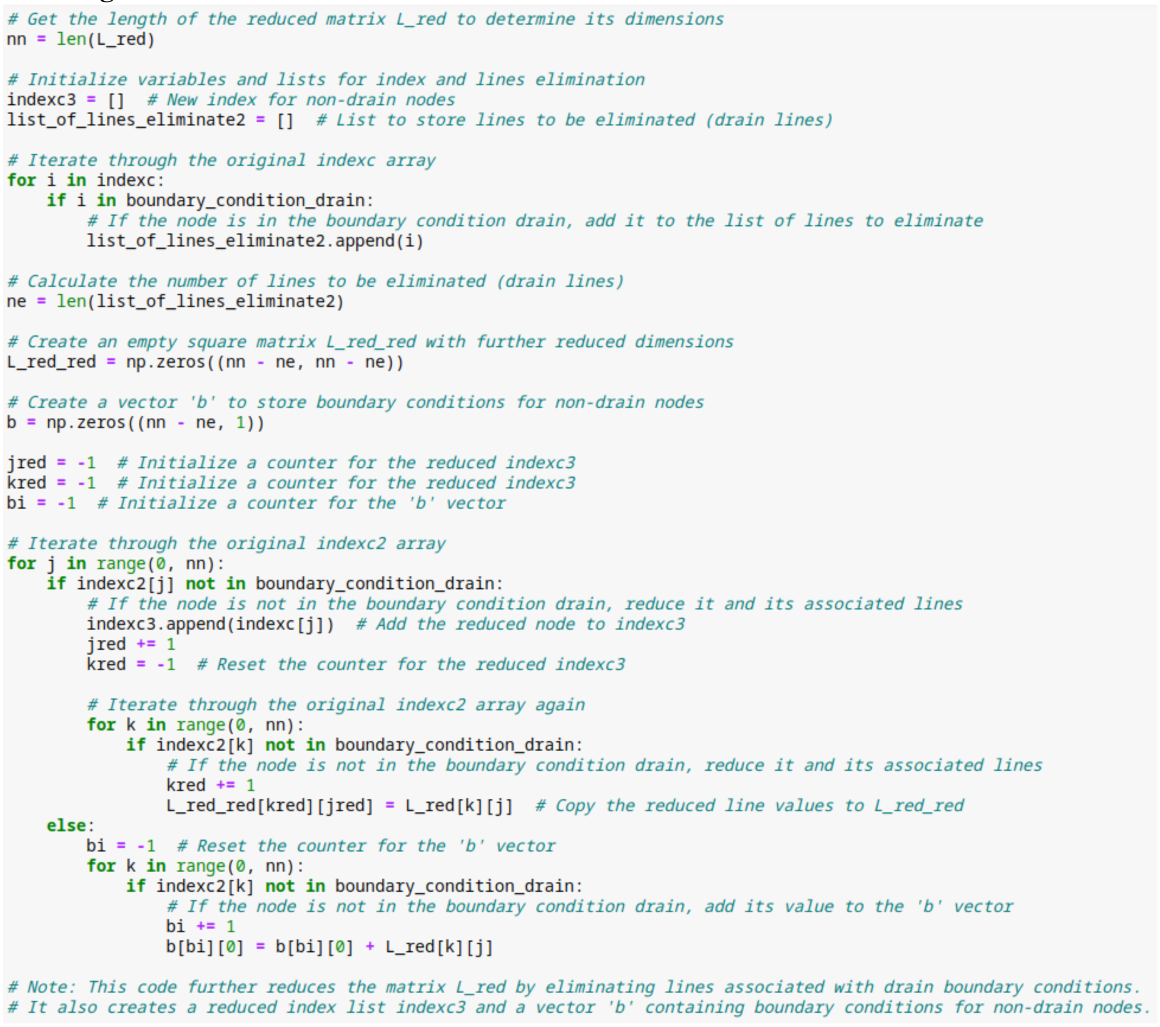
It is necessary to perform two reductions on the Laplacian matrix, with the first reduction removing the rows and columns corresponding to the source nodes. The code related to this part can be seen in Figure 3.13. The second reduction removes the drain nodes, and the code for this part can be seen in Figure 3.14. For didactic purposes, we will refer to this reduced matrix as M.

Figure 3.13 Code that does the reduction that removes the source nodes

```python
# Get the length of the matrix L to determine its dimensions
nn = len(L)

# Initialize variables and lists for index and lines elimination
indexc2 = []  # New index for non-boundary nodes
list_of_lines_eliminate = []  # List to store lines to be eliminated

# Iterate through the original indexc array
for i in indexc:
    if i in boundary_condition_source:
        # If the node is in the boundary condition source, add it to the list of lines to eliminate
        list_of_lines_eliminate.append(i)

# Calculate the number of lines to be eliminated
ne = len(list_of_lines_eliminate)

# Create an empty square matrix L_red with reduced dimensions
L_red = np.zeros((nn - ne, nn - ne))

jred = -1  # Initialize a counter for the reduced indexc2
kred = -1  # Initialize a counter for the reduced indexc2

# Iterate through the original indexc array
for j in range(0, nn):
    if indexc[j] not in boundary_condition_source:
        # If the node is not in the boundary condition source, reduce it and its associated lines
        jred += 1
        indexc2.append(indexc[j])  # Add the reduced node to indexc2
        kred = -1  # Reset the counter for the reduced indexc2

        # Iterate through the original indexc array again
        for k in range(0, nn):
            if indexc[k] not in boundary_condition_source:
                # If the node is not in the boundary condition source, reduce it and its associated lines
                kred += 1
                L_red[kred][jred] = L[k][j]  # Copy the reduced line values to L_red
```

Source: Author, 2023.

Figure 3.14 Code that does the eduction that removes the drain nodes and also create $\vec{b}$



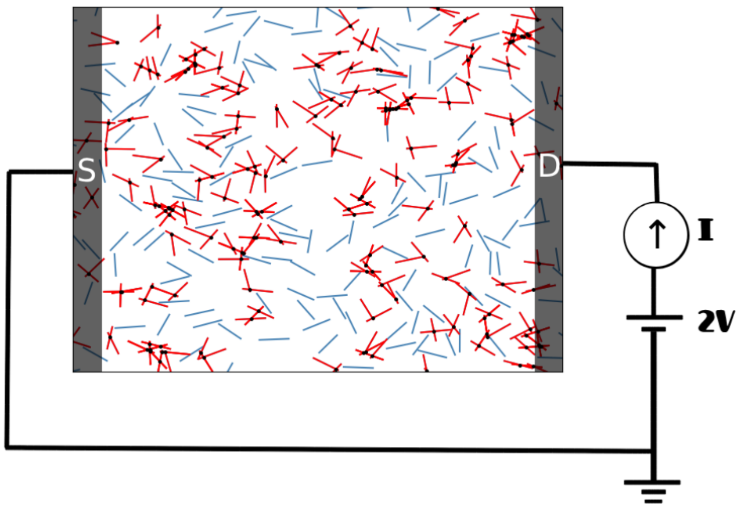
```python
# Get the length of the reduced matrix L_red to determine its dimensions
nn = len(L_red)

# Initialize variables and lists for index and lines elimination
indexc3 = []  # New index for non-drain nodes
list_of_lines_eliminate2 = []  # List to store lines to be eliminated (drain lines)

# Iterate through the original indexc array
for i in indexc:
    if i in boundary_condition_drain:
        # If the node is in the boundary condition drain, add it to the list of lines to eliminate
        list_of_lines_eliminate2.append(i)

# Calculate the number of lines to be eliminated (drain lines)
ne = len(list_of_lines_eliminate2)

# Create an empty square matrix L_red_red with further reduced dimensions
L_red_red = np.zeros((nn - ne, nn - ne))

# Create a vector 'b' to store boundary conditions for non-drain nodes
b = np.zeros((nn - ne, 1))

jred = -1  # Initialize a counter for the reduced indexc3
kred = -1  # Initialize a counter for the reduced indexc3
bi = -1  # Initialize a counter for the 'b' vector

# Iterate through the original indexc2 array
for j in range(0, nn):
    if indexc2[j] not in boundary_condition_drain:
        # If the node is not in the boundary condition drain, reduce it and its associated lines
        indexc3.append(indexc[j])  # Add the reduced node to indexc3
        jred += 1
        kred = -1  # Reset the counter for the reduced indexc3

        # Iterate through the original indexc2 array again
        for k in range(0, nn):
            if indexc2[k] not in boundary_condition_drain:
                # If the node is not in the boundary condition drain, reduce it and its associated lines
                kred += 1
                L_red_red[kred][jred] = L_red[k][j]  # Copy the reduced line values to L_red_red
    else:
        bi = -1  # Reset the counter for the 'b' vector
        for k in range(0, nn):
            if indexc2[k] not in boundary_condition_drain:
                # If the node is not in the boundary condition drain, add its value to the 'b' vector
                bi += 1
                b[bi][0] = b[bi][0] + L_red[k][j]

# Note: This code further reduces the matrix L_red by eliminating lines associated with drain boundary conditions.
# It also creates a reduced index list indexc3 and a vector 'b' containing boundary conditions for non-drain nodes.
```

Source: Author, 2023.

The reduction is required since the matrix L, constructed as described above, ends up resulting in a singular matrix. Thus, it is not possible to obtain its inverse. This happens for two reasons: the potential is only defined up to a constant value, and this constant needs to be removed. This removal is done when we eliminate from matrix L all the rows and columns associated with the ground, in other words, when we remove the source nodes. This gives us the reference potential, meaning these are the nodes that we define as having values equal to zero. This matrix remains singular since we have not established the boundary condition yet. We have not determined how to connect the network to a voltage source, which is what the vector $\vec{b}$ describes. We inject currents along specific edges into the network, and these currents are collected in $\vec{b}$ (boundary condition for Kirchhoff's current law).

Both the voltage values of the nodes that are part of the source (which are equal to zero) and the values of the nodes that are part of the drain (which are equal to the value provided by the current source) are known. To make it easier to understand this part, refer to Figure 3.15, which shows a merely illustrative example of how our numerical model works.

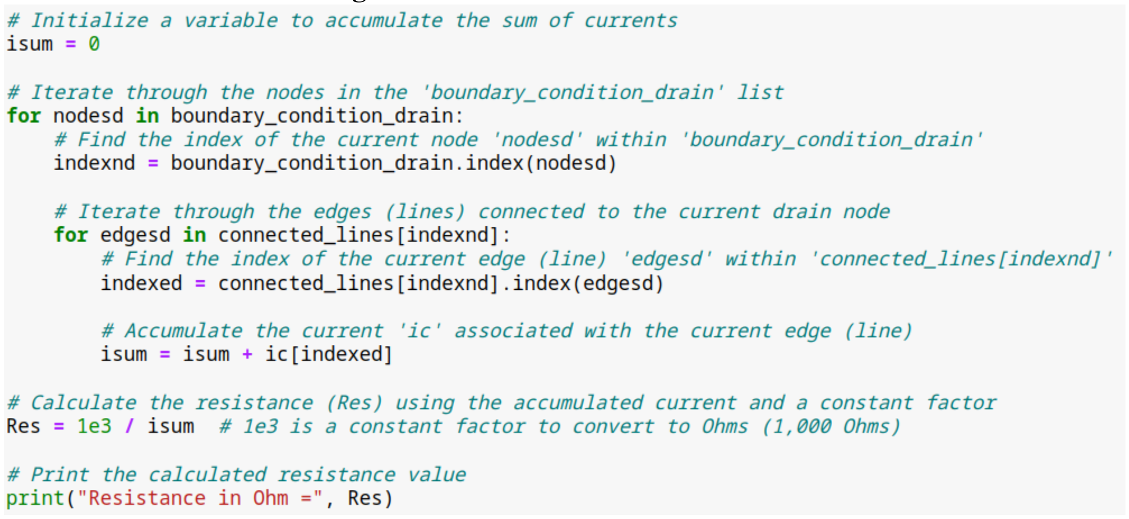
Figure 3.15 An illustrative example of how our numerical model works



Source: Author, 2023.

To calculate the voltage values of each of the internal nodes, i.e., those that are neither in the source nor in the drain, a multiplication is performed between the inverse of the matrix M and the vector $\vec{b}$. This multiplication results in a vector, which, also for didactic purposes, will be referred to as $\vec{V}$. Now, it is necessary to find the currents flowing between the line segments (sticks). For this, a multiplication is performed between matrix A and $\vec{V}$. This multiplication results in a vector I, which contains all the currents. We compute all the currents, but to compute impedance, we only add those that start flowing along an edge connected to a drain node. The impedance associated with that path is calculated by chosing a value of k, here $1\,\mathrm{k}\Omega$, and by dividing the voltage value associated with it by the current value found in the last step, as can be seen in Figure 3.16.

Figure 3.16 Calculation of the resistance



```python
# Initialize a variable to accumulate the sum of currents
isum = 0

# Iterate through the nodes in the 'boundary_condition_drain' list
for nodesd in boundary_condition_drain:
    # Find the index of the current node 'nodesd' within 'boundary_condition_drain'
    indexnd = boundary_condition_drain.index(nodesd)

    # Iterate through the edges (lines) connected to the current drain node
    for edgesd in connected_lines[indexnd]:
        # Find the index of the current edge (line) 'edgesd' within 'connected_lines[indexnd]'
        indexed = connected_lines[indexnd].index(edgesd)

        # Accumulate the current 'ic' associated with the current edge (line)
        isum = isum + ic[indexed]

# Calculate the resistance (Res) using the accumulated current and a constant factor
Res = 1e3 / isum  # 1e3 is a constant factor to convert to Ohms (1,000 Ohms)

# Print the calculated resistance value
print("Resistance in Ohm =", Res)
```

Source: Author, 2023.

To provide a practical example of impedance calculation, we will use the graph shown as an example in Figure 2.7.

First, we will present the incidence matrix in the graph, which will be our Equation (3):

$$A = \begin{bmatrix} -1 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ -1 & 0 & 0 & 1 \\ 0 & -1 & 0 & 1 \end{bmatrix} \tag{3}$$

We also need the transpose of A, which will be our Equation (4):

$$A^T = \begin{bmatrix} -1 & -1 & 0 & -1 & 0 \\ 1 & 0 & -1 & 0 & -1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{4}$$

The multiplication of A by A^T results in L, which will be our Equation (5): $L = A \cdot A^T$ where:

$$L = \begin{bmatrix} -1 & -1 & 0 & -1 & 0 \\ 1 & 0 & -1 & 0 & -1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} -1 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ -1 & 0 & 0 & 1 \\ 0 & -1 & 0 & 1 \end{bmatrix} \tag{5}$$

Therefore, matrix L is:

$$L = \begin{bmatrix} 3 & -1 & -1 & -1 \\ -1 & 3 & -1 & -1 \\ -1 & -1 & 2 & 0 \\ -1 & -1 & 0 & 2 \end{bmatrix} \tag{5}$$

Now we need to reduce matrix L. We will remove the rows and columns related to the source node (node 4) and the drain node (node 1), which will be our Equation (6).

$$M = \begin{bmatrix} 3 & -1 \\ -1 & 2 \end{bmatrix} \tag{6}$$

From this reduction, as explained earlier, we also obtain our $\vec{b}$, which will be our Equation (7):

$$\vec{b} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \tag{7}$$

Now, we will obtain vector $\vec{V}$, which will be our Equation (8). To do this, we will multiply the inverse of matrix M by vector $\vec{b}$.The multiplication of M^{-1} by $\vec{b}$ results in a $\vec{V}$ with 2 elements:

$$\vec{V} = M^{-1} \cdot \vec{b} = \begin{bmatrix} 0.4 & 0.2 \\ 0.2 & 0.6 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix} \tag{8}$$

Therefore, the result of the multiplication is the vector V:

$$\vec{V} = \begin{bmatrix} 0.6 \\ 0.8 \end{bmatrix} \tag{8}$$

This vector $\vec{V}$ also contains the voltage at the source node and the voltage at the drain node, which will be our Equation (9). Therefore,

$$\vec{V} = \begin{bmatrix} 1.0 \\ 0.6 \\ 0.8 \\ 0.0 \end{bmatrix} \tag{9}$$

Now, it is necessary to find the currents flowing between the line segments (sticks), which will be our Equation (10). For this, a multiplication is performed between matrix -A and $\vec{V}$. The multiplication of $-A$ by $\vec{V}$ results in a vector I:

$$I = -A \cdot \vec{V} = \begin{bmatrix} 1 & -1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & -1 & 0 \\ 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} 1.0 \\ 0.6 \\ 0.8 \\ 0.0 \end{bmatrix} \tag{10}$$

Per sign convention, the current always flows from the high potential to the low potential. Since in the example node 1 is the drain and node 4 is the source, the conventional current flows from 1 to 4 (and not from 4 to 1 as indirectly assumed when constructing A) and we need to change the direction of all currents. This explains the minus sign. It is most evident that the flow along edge 4 comes out negative using the difference matrix as constructed but must be positive.

We calculate all currents but add up only the ones start flow along an edge connected to a drain node. Therefore, the result of the multiplication is vector I:

$$I = \begin{bmatrix} 0.4 \\ 0.2 \\ 1.0 \\ -0.2 \\ 0.6 \end{bmatrix} \tag{10}$$

We calculate all currents but add up only the ones that start flowing along an edge connected to a drain node. The impedance associated with that path is calculated by dividing the voltage value associated with it by the current value found in the last step, which will be our Equation (11).

$$R = \frac{1000}{0.4 + 0.2 + 1} = 625.0\,\Omega \tag{11}$$

However, when dealing with the graphs generated by our model, we encountered a problem: the percolation paths resulted in a singular reduced Laplacian matrix. Therefore, it is not possible to invert it. Consequently, we cannot calculate the other values, and thus, we cannot calculate the impedance associated with the percolation path. The reason why the percolation paths are not working is that there are 'dead-end points' (dangling internal sticks), i.e., internal nodes (neither source nor drain) that are connected to the path only by a single edge. Current cannot flow through these nodes, and therefore, their potential is 'floating', leading to a singular matrix. The percolation graph needs to be pruned. However, eliminating each of these dangling sticks points can create more dangling sticks. Therefore, this process needs to continue until all internal dangling sticks are eliminated. This task is left for future work.

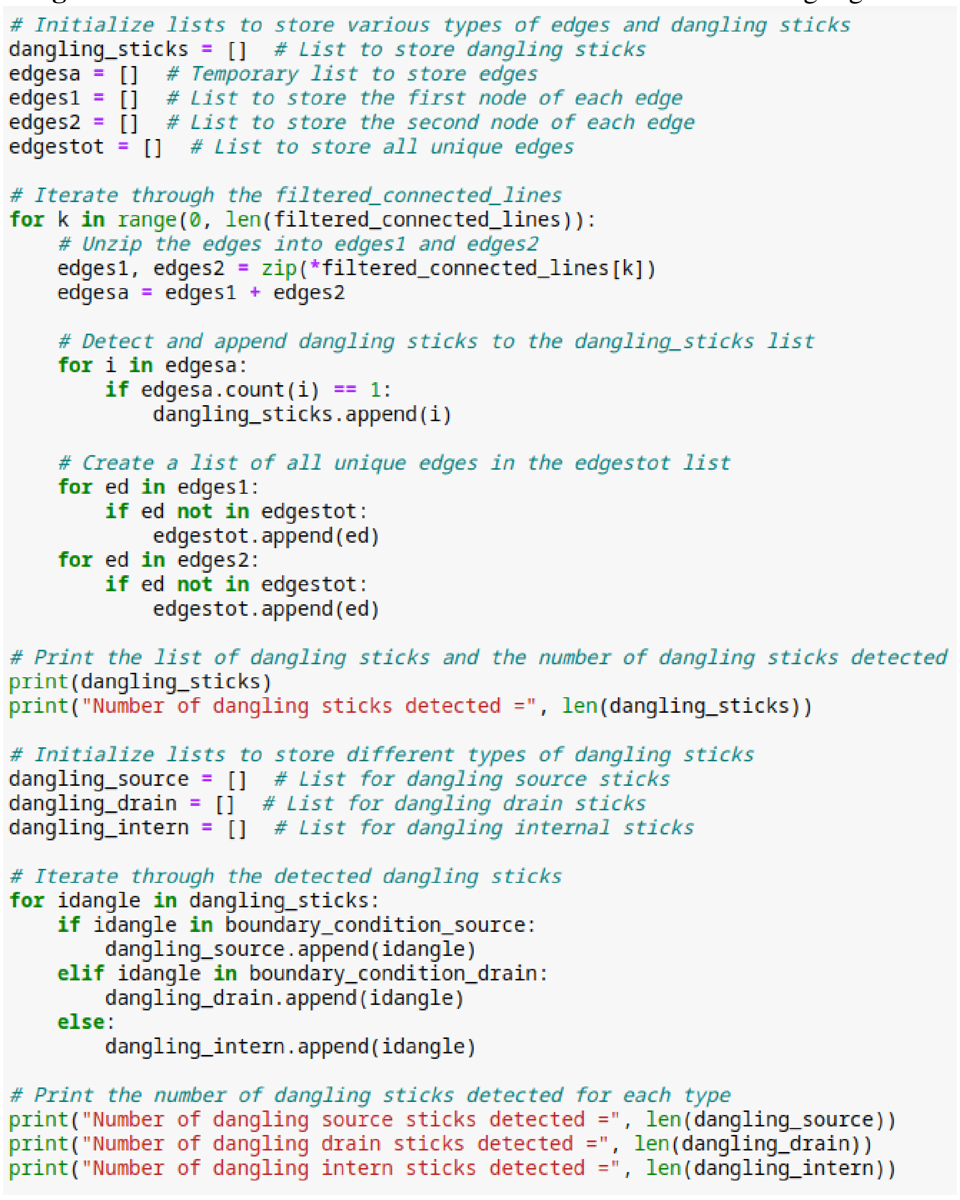
Since the pruning of the percolation graphs has not yet been implemented, the computation of the impedance for the shown examples in Chapter 4 is not possible. However, pruned graphs can be passed to the impedance module of the software by manually generating the graph description. For example, the graph we just calculated the impedance manually, can be defined as:

```
connected_nodes = [1,2,3,4]
connected_lines = [(1,2,4),(1,3,5),(2,3),(4,5)]
boundary_condition_drain = [1]
boundary_condition_source = [4]
```

Passing this graph to the impedance module, we obtain an impedance of 625 Ω.

Even though this problem was not resolved in this project, we attempted to understand it a bit more. For this purpose, we conducted an analysis of these dangling sticks. To perform this analysis, we implemented a function, as can be seen in Figure 3.17, designed to check the number of dangling source sticks, dangling drain sticks, and dangling internal sticks.

Figure 3.17 Code of the function that checks the existence of Dangling Sticks



```python
# Initialize lists to store various types of edges and dangling sticks
dangling_sticks = []  # List to store dangling sticks
edgesa = []  # Temporary list to store edges
edges1 = []  # List to store the first node of each edge
edges2 = []  # List to store the second node of each edge
edgestot = []  # List to store all unique edges

# Iterate through the filtered_connected_lines
for k in range(0, len(filtered_connected_lines)):
    # Unzip the edges into edges1 and edges2
    edges1, edges2 = zip(*filtered_connected_lines[k])
    edgesa = edges1 + edges2

    # Detect and append dangling sticks to the dangling_sticks list
    for i in edgesa:
        if edgesa.count(i) == 1:
            dangling_sticks.append(i)

    # Create a list of all unique edges in the edgestot list
    for ed in edges1:
        if ed not in edgestot:
            edgestot.append(ed)
    for ed in edges2:
        if ed not in edgestot:
            edgestot.append(ed)

# Print the list of dangling sticks and the number of dangling sticks detected
print(dangling_sticks)
print("Number of dangling sticks detected =", len(dangling_sticks))

# Initialize lists to store different types of dangling sticks
dangling_source = []  # List for dangling source sticks
dangling_drain = []  # List for dangling drain sticks
dangling_intern = []  # List for dangling internal sticks

# Iterate through the detected dangling sticks
for idangle in dangling_sticks:
    if idangle in boundary_condition_source:
        dangling_source.append(idangle)
    elif idangle in boundary_condition_drain:
        dangling_drain.append(idangle)
    else:
        dangling_intern.append(idangle)

# Print the number of dangling sticks detected for each type
print("Number of dangling source sticks detected =", len(dangling_source))
print("Number of dangling drain sticks detected =", len(dangling_drain))
print("Number of dangling intern sticks detected =", len(dangling_intern))
```

Source: Author, 2023.

The function operates as follows: it processes a list of connected line segments represented as tuples in the 'filtered connected lines' list. This list specifically contains the lines that are part of the percolation path. Within these line segments, the function identifies and categorizes 'dangling sticks,' which are line segments that are not fully connected to other line segments.

The code iterates through each tuple in the 'filtered connected lines', extracts its constituent line segments, and determines which line segments are dangling by checking their connectivity within the list. Furthermore, it categorizes these dangling sticks into three distinct groups based on predefined boundary conditions: source, drain, and internal. As part of its functionality, the code also prints the detected dangling sticks along with the count of each category.

CHAPTER 4

Experiments and Results

To analyze how the model developed in this project operates and, as a result, gain a better understanding of Random Carbon Nanotube Networks for Flexible Electronics, we will conduct experiments. In these experiments, we will vary two key parameters: the density, represented by the number of lines N, and the length of line segments a.

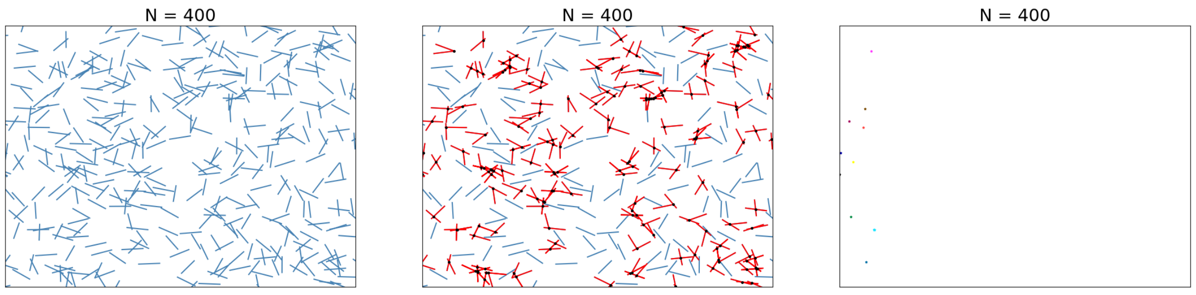
The objective of these experiments is to estimate the specific values of N and a at which our model begins to generate Random Networks that exhibit percolation paths. This investigation will provide valuable insights into the behavior of the model and its ability to simulate networks conducive to percolation phenomena.

4.1 Experiments with a fixed stick length

The first experiments to be conducted will be carried out with a fixed value of a equal to 0.06 and varying the value of N.

For $N = 400$, we have in Figure 4.1 respectively the plots of the RLG, the identified intersections, and the found paths.

Figure 4.1 Networks generated for $N = 400$

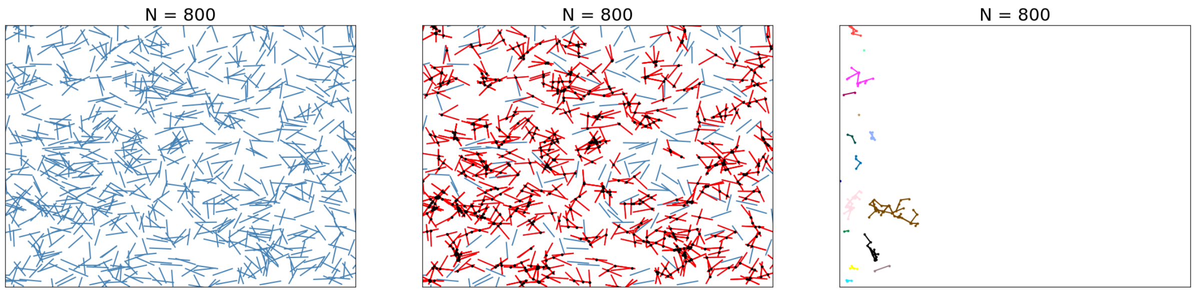


Source: Author, 2023.

Checking the array that contains the intersection points, it can be noted that, for the value of $N = 400$, there were 183 intersections, which is a significant number, as it is nearly half the number of lines. Nevertheless, no percolation path was found. In this case, not even a non-percolation path was identified. Only some isolated nodes were shown as possible paths in the graph.

For $N = 800$, we have in Figure 4.2 respectively the plots of the RLG, the identified intersections, and the found paths.

Figure 4.2 Graphs generated for $N = 800$

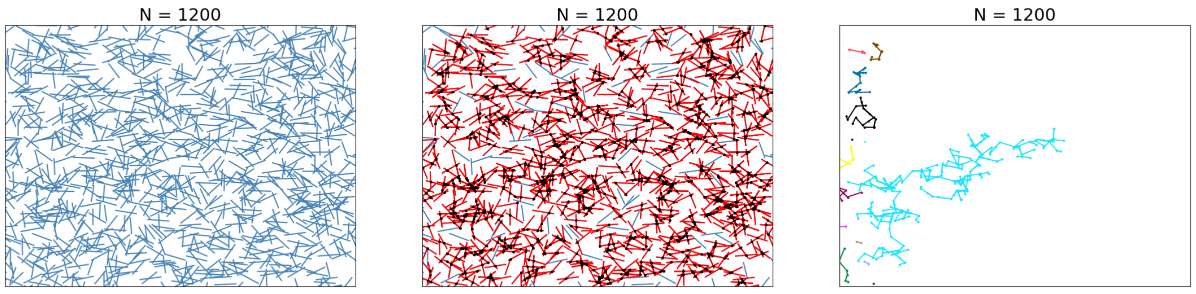


Source: Author, 2023.

Checking the array that contains the intersection points, it can be observed that for the value of $N = 800$, 712 intersections occurred, which is quite significant, as it is very close to the number of lines. Nevertheless, no percolation path was found in this case. Instead, some non-percolation paths were identified.

For $N = 1200$, we have in the figure 4.3 respectively the plots of the RLG, the identified intersections, and the found paths.

Figure 4.3 Graphs generated for $N = 1200$



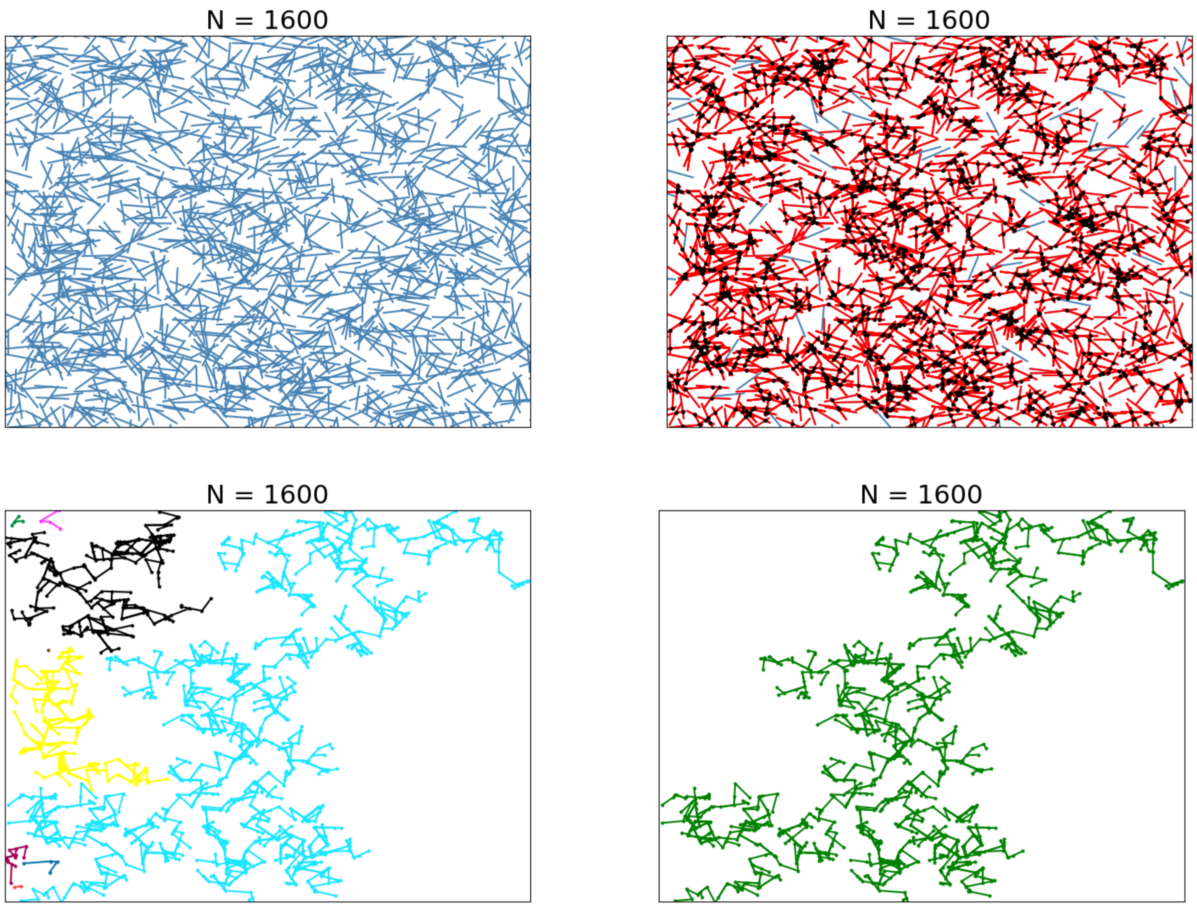
Source: Author, 2023.

Checking the array containing the intersection points, it can be noted that, for the value of $N = 1200$, there were 1505 intersections, which is quite interesting, as this number is greater than the number of lines in the RLG. Nevertheless, even so, no percolation paths were found.

However, in this case, it can be observed that non-percolation paths were found, and some of them are longer than when $N = 800$.

For $N = 1600$, we have in the figure 4.4 respectively the plots of the RLG, the identified intersections, all the found paths, and the percolating path.

Figure 4.4 Graphs generated for $N = 1600$

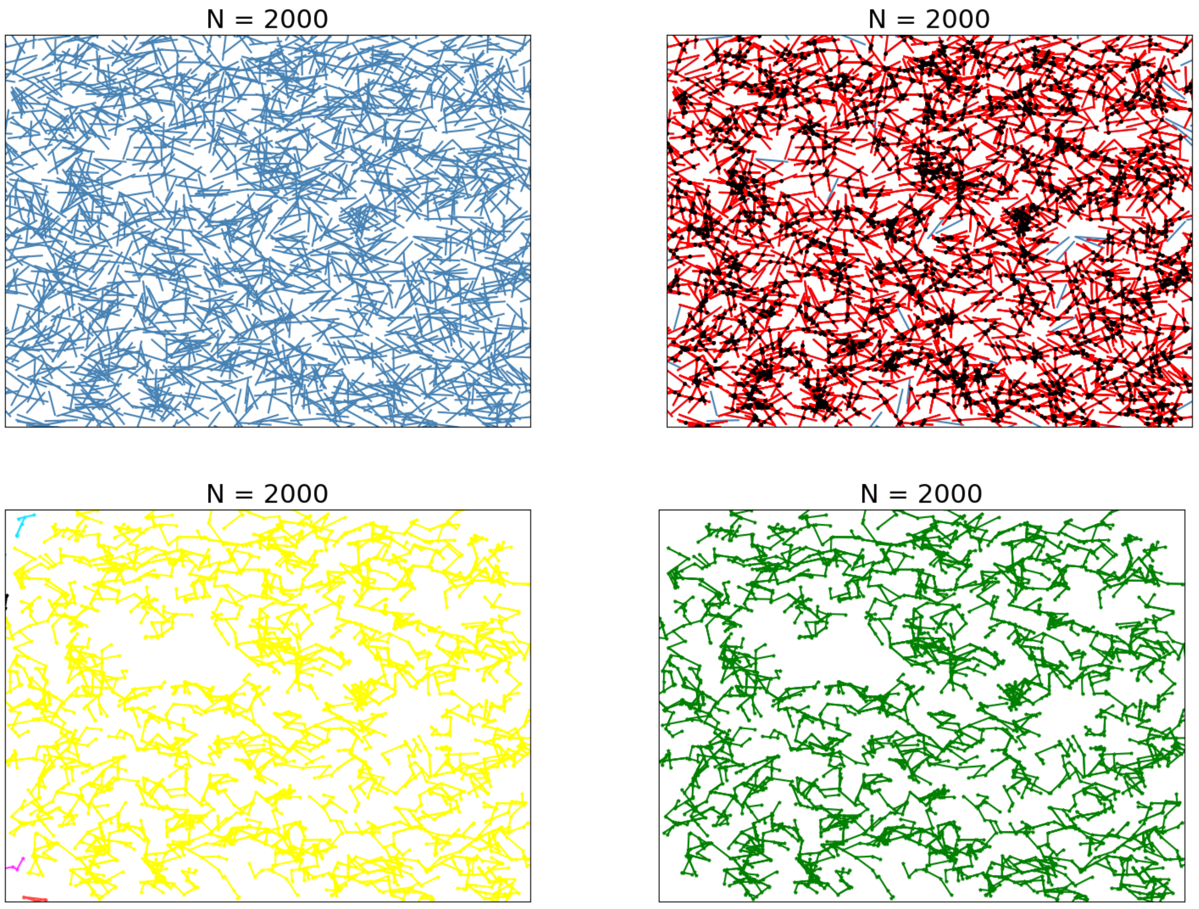


Source: Author, 2023.

Checking the array containing intersection points, it can be observed that, for $N = 1600$, there were 2718 intersections. In this experiment, it can be observed that fewer non-percolation paths were found, but the ones that were found are longer than those found in previous experiments. Furthermore, it was possible to find our first percolation path.

For $N = 2000$, we have in the figure 4.5 respectively the plots of the RLG, the identified intersections, all the found paths, and the percolating path.

Figure 4.5 Graphs generated for $N = 2000$



Source: Author, 2023.

Checking the array containing the intersection points, it can be observed that, for $N = 2000$, there were 4364 intersections, which is something interesting, as this number is more than double the number of lines in the RLG. This time, it was also possible to find a percolation path, but only a single path, which is longer than the one found in the previous experiment. It can be observed that, instead of forming two or more different percolation paths, the line segments (sticks) tend to connect and form a single percolation path.

The experiments conducted offer several noteworthy observations and conclusions.

Initially, it becomes apparent that as the number of lines N increases, the density of intersections also rises significantly. For instance, when N reaches 2000, the number of intersections surpasses double the number of lines in the network, indicating a dense crisscross of line segments.

Surprisingly, these experiments reveal that the mere presence of intersections does not guarantee the existence of percolation paths. Even with a substantial number of intersections, percolation paths are not evident until a critical threshold is reached, which in this case is $N = 1600$.

Additionally, the length of line segments a seems to impact the characteristics of non-percolation paths, with longer segments leading to more extended non-percolation paths.

Finally, it is intriguing to note that, rather than multiple percolation paths, the line segments tend to connect and form a single, longer percolation path as the density of intersections increases. These findings collectively emphasize the critical role of both N and a in the formation of percolation networks and provide valuable insights into the behavior of Random Carbon Nanotube Networks for Flexible Electronics under varying conditions.

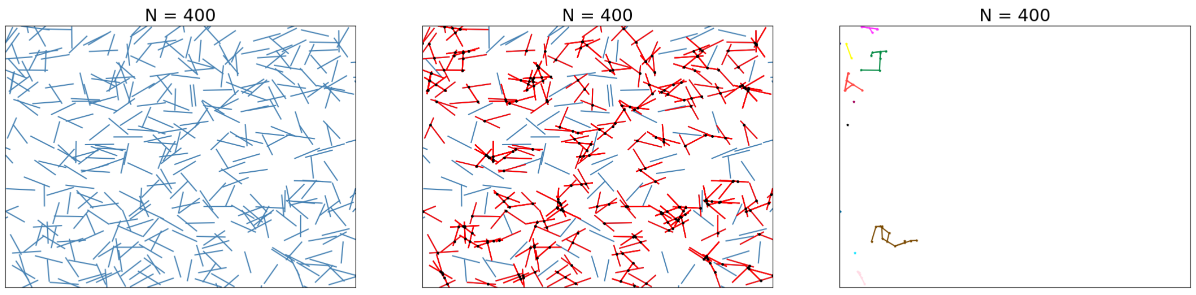
4.2 Experiments with a fixed number of sticks

Now we will conduct experiments with a fixed value of N and vary the value of a.

In the first experiment, we will fix N at 400 and vary the value of a.

For $a = 0.8$, we have in Figure 4.6 respectively the plots of the RLG, the identified intersections, and all the found paths.

Figure 4.6 Graphs generated for $N = 400$ and $a = 0.8$

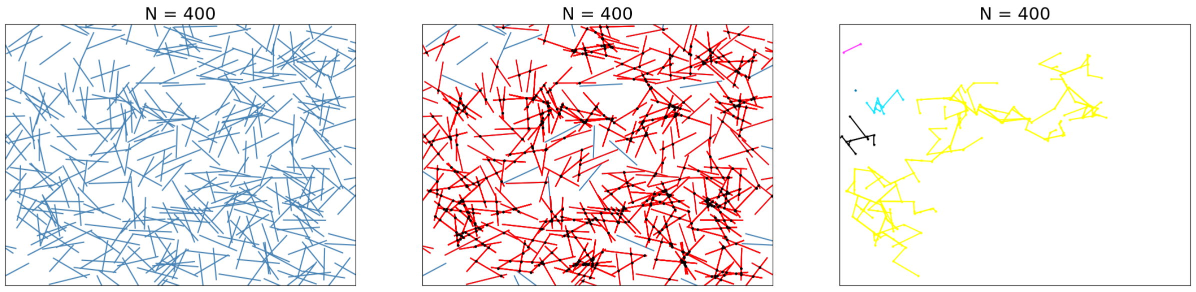


Source: Author, 2023.

Checking the array containing the intersection points, it can be observed that, for the value of $a = 0.08$, there were 288 intersections, which is quite interesting, as there are more than 100 intersections more than when the experiment with this number of line segments was executed with $a = 0.06$. Furthermore, it can be observed that when the experiment for $N = 400$ was executed with the value of $a = 0.06$, only isolated points were found, and this time it was already possible to find some non-percolation paths.

For $a = 0.12$, we have in Figure 4.7 respectively the plots of the RLG, the identified intersections, and all the found paths.

Figure 4.7 Graphs generated for $N = 400$ and $a = 0.12$

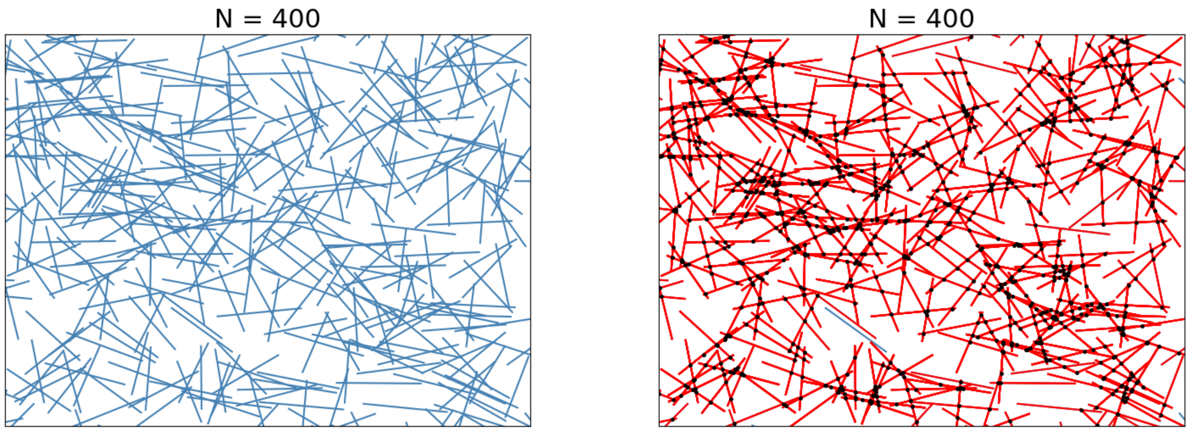


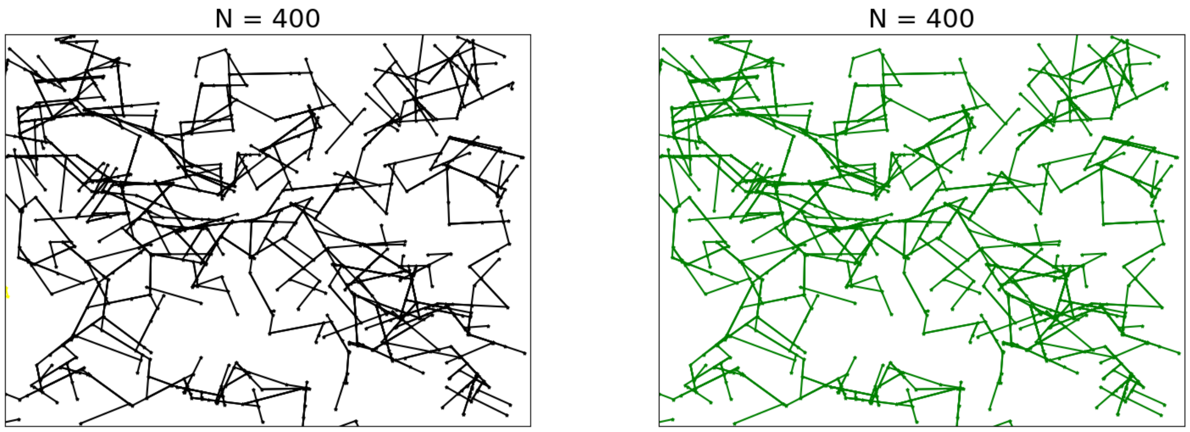
Source: Author, 2023.

Checking the array that contains the intersection points, it can be observed that, for the value of $a = 0.12$, there were 641 intersections, which is quite interesting, as the number of intersections is greater than the number of line segments in the RLG. No percolation path was found, but at least one non-percolation path can already be observed, which is considerably long.

For $a = 0.16$, we have in Figure 4.8 respectively the plots of the RLG, the identified intersections, all the found paths, and the percolating path found.

Figure 4.8 Graphs generated for $N = 400$ and $a = 0.16$





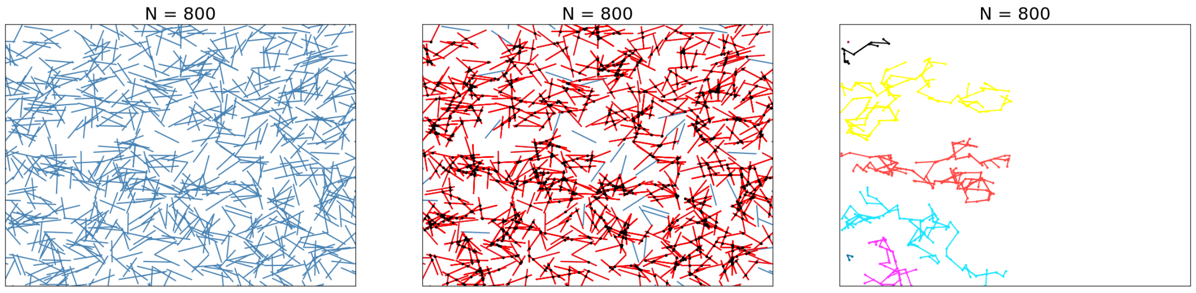
Source: Author, 2023.

Checking the array that contains the intersection points, one can observe that, for the value of $a = 0.16$, there were 1110 intersections, which is quite fascinating since the number of intersections is nearly three times greater than the number of line segments in the RLG. An intriguing aspect to note in the intersection graph is that almost all line segments intersected, and some of them intersected with more than one line. Furthermore, only a single path was discovered, and it happens to be the percolation path.

In the second experiment, we will fix N at 800 and vary the value of a.

For $a = 0.08$, we have in Figure 4.9 respectively the plots of the RLG, the identified intersections, and all the found paths.

Figure 4.9 Graphs generated for $N = 800$ and $a = 0.08$

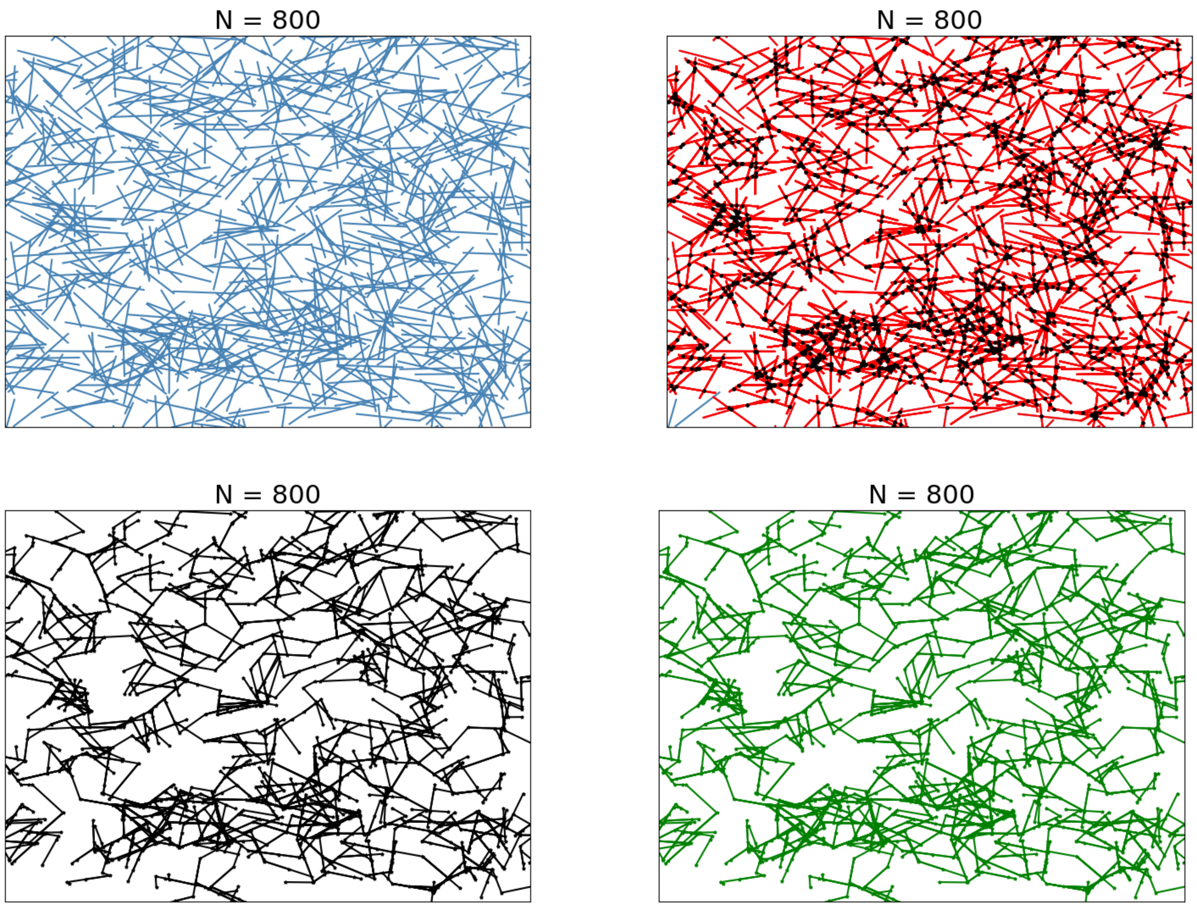


Source: Author, 2023.

Checking the array containing intersection points, it can be observed that, for the value of $a = 0.08$, there were 1240 intersections. When observing the intersection graph, one can notice that only a few line segments (sticks) did not intersect. No percolation path was found, but it can be observed that at least three of the non-percolation paths reach approximately halfway into the unit square's area.

For $a = 0.12$, we have in Figure 4.10 respectively the plots of the RLG, the identified intersections, all the found paths, and the percolating path found.

Figure 4.10 Graphs generated for $N = 800$ and $a = 0.12$



Source: Author, 2023.

Checking the set containing the intersection points, it can be observed that, for the value of $N = 800$ and $a = 0.12$, there were 2598 intersections, which is something interesting, as this number is more than three times the number of lines in the RLG. Furthermore, it can be observed that we found a single path, which is precisely the percolation path.

The experiments conducted offer several noteworthy observations and conclusions.

Initially, by varying the length of line segments a while keeping the number of lines N fixed, it becomes evident that increasing the value of a results in a higher density of intersections. For instance, at $a = 0.16$, the number of intersections nearly triples, emphasizing the significant impact of a on network connectivity. Importantly, these experiments demonstrate that as a increases, non-percolation paths become more prevalent and notably longer. Furthermore, the presence of a percolation path is only observed at specific values of *a*, indicating a critical threshold for network formation.

Additionally, when keeping *N* constant and varying *a*, it is remarkable that the number of intersections can exceed three times the number of lines in the network, underlining the complex nature of these networks. Most notably, a single, extended percolation path is discovered at certain *a* values, indicating the existence of a critical *a* for percolation formation.

Overall, these experiments reveal that both *N* and *a* play pivotal roles in the emergence of percolation networks in Random Carbon Nanotube Networks for Flexible Electronics. The findings offer crucial insights into the network's behavior under changing conditions and provide valuable guidance for optimizing such networks for practical applications.

To determine the critical values of *N* and *a* for percolation, it is necessary to conduct many more experiments. For example, we can perform 100 or more experiments with the same values of *N* and *a* and check how often these same *N* and *a* values result in percolation paths. This will provide us with sufficient data to calculate statistical information, such as the mean, median, and standard deviation of these values. This more detailed investigation will be left for future work when the pruning of percolation graphs is also implemented.

4.3 Impedance Computation

To show the completeness of the installed modules for the impedance computation, a series of small pruned graphs were generated manually and their impedance assessed. Here a summary of the obtained results:

connected_nodes = [1, 2, 3, 4, 5]
connected_lines = [(1, 2, 4), (1, 3, 5), (2, 3, 1), (3, 4, 3), (4, 5, 2)]
boundary_condition_drain = [2, 5]
boundary_condition_source = [1, 4]

Passing this graph to the impedance module, we obtain an impedance of 300 Ω.

connected_nodes = [1, 2, 3, 4, 5, 6]
connected_lines = [(1, 2, 4), (1, 3, 5), (2, 3, 1), (3, 4, 3), (4, 5, 2), (5, 6, 1)]
boundary_condition_drain = [2, 4]
boundary_condition_source = [1, 5]

Passing this graph to the impedance module, we obtain an impedance of 2000 Ω.

connected_nodes = [1, 2, 3, 4]
connected_lines = [(1, 2, 4), (1, 3, 5), (2, 3, 1), (3, 4, 3)]
boundary_condition_drain = [4]
boundary_condition_source = [2]

Passing this graph to the impedance module, we obtain an impedance of 4999.99 Ω.

connected_nodes = [1, 2, 3, 4]
connected_lines = [(1, 2, 4), (1, 3, 5), (2, 3, 1), (3, 4, 3)]
boundary_condition_drain = [3]
boundary_condition_source = [2]

Passing this graph to the impedance module, we obtain an impedance of 454.54 Ω.

connected_nodes = [1, 2, 3]
connected_lines = [(1, 2, 2), (2, 3, 3)]
boundary_condition_drain = [3]
boundary_condition_source = [1]

Passing this graph to the impedance module, we obtain an impedance of 833.33 Ω.

connected_nodes = [1, 2, 3, 4, 5, 6, 7, 8, 9]
connected_lines = [(1, 2, 2), (1, 3, 3), (2, 3, 1), (3, 4, 4), (4, 5, 2), (5, 6, 3), (6, 7, 1), (7, 8, 2), (8, 9, 3)]
boundary_condition_drain = [6, 7, 8]
boundary_condition_source = [1, 2, 3]

Passing this graph to the impedance module, we obtain an impedance of 129.03 Ω.

CHAPTER 5

Conclusion and Future Work

In conclusion, this project has delved into the realm of flexible electronics, specifically focusing on the Random-Line-Graph (RLG) model. Through a series of insightful experiments, we have gained valuable insights into the factors governing the formation of percolation paths within the Random Networks of Carbon Nanotube for flexible electronics.

The experiments conducted in this study illuminate the intricate structure governing Random Carbon Nanotube Networks for Flexible Electronics. The findings underscore the pivotal roles of both the number of lines N and the length of line segments a in shaping the formation and characteristics of percolation networks within these systems.

As N increases, the density of intersections escalates, reflecting the network's growing complexity. Simultaneously, specific values of N are associated with the emergence of percolation paths, highlighting the existence of a critical threshold for network formation. Similarly, a profoundly influences network connectivity, with certain a values leading to longer non-percolation paths. Importantly, these experiments also reveal that percolation paths are observed at specific values of a, indicating the presence of a critical a for percolation formation.

These insights provide crucial guidance for optimizing Random Carbon Nanotube Networks for flexible electronics, offering a deeper understanding of their behavior under changing conditions and the significance of both N and a in percolation path formation.

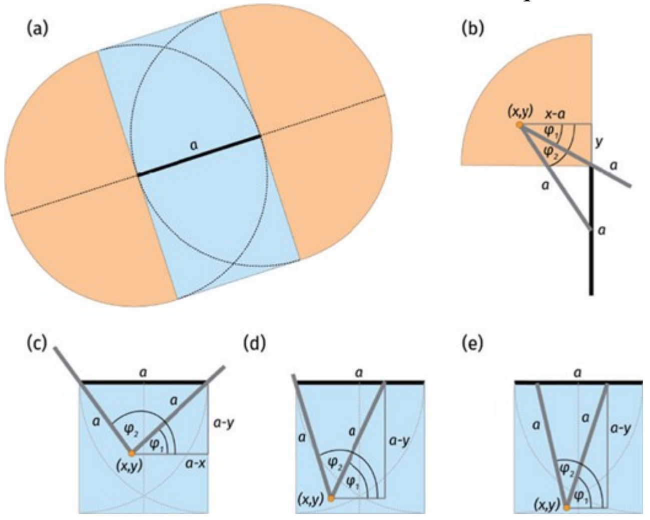
Looking forward, future research should focus on addressing the challenge of eliminating dangling sticks in the RLG, which is crucial for accurate impedance calculations. Furthermore, some specific improvements can be made to the model of this project. For example, the conductance of the nanotube that forms the percolation path, instead of being a constant value, should be calculated, as this will provide a more accurate result. Optimizations in some functions, such as the one that searches for the path from a node, should also be performed, as this will allow the software to deliver results more quickly. Another thing that can be done is to create a GUI for the software and make it open source, as commercial available CAD software has very expensive licenses. One last suggestion is the development of a model for thin-film transistors based on CNTs' RNs.

CHAPTER 6

Appendix

6.1 Regions

Figure 6.1 Possible area of the intersection divided into squares and quarter circles



Fonte: BÖTTCHER, 2020, p. 8.

In the article 'A Random-Line-Graph Approach to Overlapping Line Segments' (Böttcher, 2020), the unit square area is divided into squares and quarter circles (see Figure 6.1), and it is stated that two segments can only intersect if they are both in the same section of the area where the unit square was subdivided. Furthermore, only the x-range, the formula for y, and the angle θ were defined for 1/4 of each of the regions.

In this project, the unit square area was divided in the same way. However, in order to carry out the simulation, testing, and reproduction of the results presented in the cited article - that is, to find intersections and thus a percolation path - it was necessary to mathematically obtain the corresponding values of x, y, and θ for each of the other sections of the area. All the found values are listed below:

Region I
To the left-lens-shaped corner of the lower square-shaped region:

$$x \in [0, \frac{a}{2}]$$
$$y \in [a - \sqrt{a^2 - (x-a)^2}, a]$$
$$\theta \in [\arctan\left(\frac{a-y}{a-x}\right), \frac{\pi}{2} + \arctan\left(\frac{x}{a-y}\right)]$$

To the right-lens-shaped corner of the lower square-shaped region:

$$x \in [\frac{a}{2}, a]$$
$$y \in [a - \sqrt{a^2 - x^2}, a]$$
$$\theta \in [\arctan\left(\frac{a-y}{a-x}\right), \frac{\pi}{2} + \arctan\left(\frac{x}{a-y}\right)]$$

To the left-lens-shaped corner of the upper square-shaped region:

$$x \in [0, \frac{a}{2}]$$
$$y \in [a, a + \sqrt{a^2 - (x-a)^2}]$$
$$\theta \in \left[\pi + \arctan\left(\frac{y-a}{x}\right), \frac{3\pi}{2} + \arctan\left(\frac{a-x}{y-a}\right)\right]$$

To the right-lens-shaped corner of the upper square-shaped region:

$$x \in [\frac{a}{2}, a]$$
$$y \in [a, a + \sqrt{a^2 - x^2}]$$
$$\theta \in \left[\pi + \arctan\left(\frac{y-a}{x}\right), \frac{3\pi}{2} + \arctan\left(\frac{a-x}{y-a}\right)\right]$$

Region II
To the lower-left corner of the square-shaped region, section Inferior I, we have:

$$x \in [0, \frac{a}{2}]$$
$$y \in [a - \sqrt{a^2 - x^2}, a - \sqrt{a^2 - (x-a)^2}]$$
$$\theta \in [\arcsin\left(\frac{a-y}{a}\right), \frac{\pi}{2} + \arctan\left(\frac{x}{a-y}\right)]$$

To the lower-right corner of the square-shaped region, section Inferior II, we have:

$$x \in [\frac{a}{2}, a]$$
$$y \in [a - \sqrt{a^2 - (x-a)^2}, a - \sqrt{a^2 - x^2}]$$
$$\theta \in [\arctan\left(\frac{a-y}{a-x}\right), \frac{\pi}{2} + \arccos\left(\frac{a-y}{a}\right)]$$

To the upper-left corner of the square-shaped region, section Superior I, we have:

$$x \in [0, \frac{a}{2}]$$
$$y \in [a + \sqrt{a^2 - (x-a)^2}, a + \sqrt{a^2 - x^2}]$$
$$\theta \in [\pi + \arctan\left(\frac{y-a}{x}\right), \frac{3\pi}{2} + \arccos\left(\frac{y-a}{a}\right)]$$

To the upper-right corner of the square-shaped region, section Superior II, we have:

$$x \in \left[\frac{a}{2}, a\right]$$
$$y \in \left[a + \sqrt{a^2 - x^2}, a + \sqrt{a^2 - (x-a)^2}\right]$$
$$\theta \in [\pi + \arcsin\left(\frac{y-a}{a}\right), \frac{3\pi}{2} + \arctan\left(\frac{a-x}{y-a}\right)]$$

Region III
To the bottom-left corner of the lower square-shaped region:

$$x \in \left[0, \frac{a}{2}\right]$$
$$y \in \left[0, a - \sqrt{a^2 - x^2}\right]$$
$$\theta \in [\arcsin\left(\frac{a-y}{a}\right), \frac{\pi}{2} + \arccos\left(\frac{a-y}{a}\right)]$$

To the bottom-right corner of the lower square-shaped region:

$$x \in \left[\frac{a}{2}, a\right]$$
$$y \in \left[0, a - \sqrt{a^2 - (x-a)^2}\right]$$
$$\theta \in [\arcsin\left(\frac{a-y}{a}\right), \frac{\pi}{2} + \arccos\left(\frac{a-y}{a}\right)]$$

To the top-left corner of the upper square-shaped region:

$$x \in [0, \frac{a}{2}]$$
$$y \in [a, a + \sqrt{a^2 - x^2}]$$
$$\theta \in [\pi + \arcsin\left(\frac{y-a}{a}\right), \frac{3\pi}{2} + \arccos\left(\frac{y-a}{a}\right)]$$

To the top-right corner of the upper square-shaped region:

$$x \in [\frac{a}{2}, a]$$
$$y \in [a, a + \sqrt{a^2 - x^2}]$$
$$\theta \in [\pi + \arcsin\left(\frac{y-a}{a}\right), \frac{3\pi}{2} + \arccos\left(\frac{y-a}{a}\right)]$$

Region IV
To the lower-left quarter of the left circle:

$$x \in [-a, 0]$$
$$y \in [a - \sqrt{a^2 - x^2}, a]$$
$$\theta \in [\arcsin\left(\frac{a-y}{a}\right), \arctan\left(\frac{a-y}{0-x}\right)]$$

To the lower-right quarter of the right circle:

$$x \in [a, 2a]$$
$$y \in [a - \sqrt{a^2 - (x-a)^2}, a]$$
$$\theta \in [\pi - \arctan\left(\frac{a-y}{x-a}\right), \pi - \arcsin\left(\frac{a-y}{a}\right)]$$

To the upper-left quarter of the left circle:

$$x \in [-a, 0]$$
$$y \in [a, a + \sqrt{a^2 - x^2}]$$
$$\theta \in [\frac{3\pi}{2} + \arctan\left(\frac{0-x}{y-a}\right), \frac{3\pi}{2} + \arccos\left(\frac{y-a}{a}\right)]$$

To the upper-right quarter of the right circle:

$$x \in [2a, a]$$
$$y \in [a, a + \sqrt{a^2 - (x-a)^2}]$$
$$\theta \in [\frac{3\pi}{2} - \arccos\left(\frac{y-a}{a}\right), \frac{3\pi}{2} - \arctan\left(\frac{x-a}{y-a}\right)]$$

6.2 Stick-Percolation Model

This is a link to a repository on GitHub that contains the model developed in this project: Stick-Percolation Model

Bibliography

[1] Derya Baran, Daniel Corzo, and Guillermo Blazquez. “Flexible Electronics: Status, Challenges and Opportunities”. In: *Frontiers in Electronics* 1 (2020). ISSN: 2673-5857. DOI: 10.3389/felec.2020.594003. URL: https://www.frontiersin.org/articles/10.3389/felec.2020.594003.

[2] Zixian Wang et al. “Flexible Electronics and Healthcare Applications”. In: *Frontiers in Nanotechnology* 3 (2021). ISSN: 2673-3013. DOI: 10.3389/fnano.2021.625989. URL: https://www.frontiersin.org/articles/10.3389/fnano.2021.625989.

[3] Maria Smolander and Liisa Hakola. *VTT study: Environmental impact of flexible electronics can be reduced by almost 90%*. Visited: 30/08/2023. 2023. URL: https://www.vttresearch.com/en/news-and-ideas/vtt-study-environmental-impact-flexible-electronics-can-be-reduced-almost-90.

[4] YongAn Huang, YeWang Su, and Shan Jiang. “Structural Engineering of Flexible Electronics”. In: *Flexible Electronics: Theory and Method of Structural Design*. Singapore: Springer Nature Singapore, 2022, pp. 1–26. ISBN: 978-981-19-6623-1. DOI: 10.1007/978-981-19-6623-1_1.

[5] Arm Ltd. *What is Electronic Design Automation (EDA)?* Visited: 30/08/2023. 2023. URL: https://www.arm.com/glossary/electronic-design-automation#:~:text=Electronic%20Design%20Automation%20(EDA)%20is,boards%2C%20integrated%20circuits%20and%20microprocessors..

[6] Phil Kinnane. *What Is ECAD? Why is It Used in FEA?* Visited: 30/08/2023. 2012. URL: https://www.comsol.com/blogs/what-is-ecad-why-is-it-used-in-fea/.

[7] Tsung-Ching Huang et al. “Process Design Kit and Design Automation for Flexible Hybrid Electronics”. In: *International Symposium on VLSI Design, Automation and Test (VLSI-DAT)* (2019), pp. 1–2. DOI: 10.1109/VLSI-DAT.2019.8741745.

[8] V. K. Sangwan et al. “Optimizing transistor performance of percolating carbon nanotube networks”. In: *Applied Physics Letters* 97 (2010), p. 043111. DOI: 10.1063/1.3469930. URL: https://doi.org/10.1063/1.3469930.

[9] Ho-Kyun Jang et al. “Electrical percolation thresholds of semiconducting single-walled carbon nanotube networks in field-effect transistors”. In: *Phys. Chem. Chem. Phys.* 17 (2015), pp. 6874–6880. DOI: 10.1039/C4CP05964F.

[10] Rama Dubey et al. "Functionalized carbon nanotubes: synthesis, properties and applications in water purification, drug delivery, and material and biomedical sciences". In: *Nanoscale Adv.* 3 (2021), pp. 5722–5744. DOI: 10.1039/D1NA00293G.

[11] Mohd Nurazzi Norizan et al. "Carbon nanotubes: functionalisation and their application in chemical sensors". In: *RSC Adv.* 10 (2020), pp. 43704–43732. DOI: 10.1039/D0RA09438B.

[12] K. C. Sivaganga and Titto Varughese. "Physical Properties of Carbon Nanotubes". In: *Handbook of Carbon Nanotubes*. Ed. by Jiji Abraham, Sabu Thomas, and Nandakumar Kalarikkal. Cham: Springer International Publishing, 2022, pp. 283–297.

[13] Lucas Böttcher. "A Random-Line-Graph Approach to Overlapping Line Segments". In: *Journal of Complex Networks* 8.4 (2020). ISSN: 2051-1329. DOI: 10.1093/comnet/cnaa029.

[14] Nicolas F. Zorn and Jana Zaumseil. "Charge transport in semiconducting carbon nanotube networks". In: *Applied Physics Reviews* 8.4 (2021), p. 041318. ISSN: 1931-9401. DOI: 10.1063/5.0065730.

[15] C.-W. Nan, Y. Shen, and Jing Ma. "Physical Properties of Composites Near Percolation". In: *Annual Review of Materials Research* 40.1 (2010), pp. 131–151. DOI: 10.1146/annurev-matsci-070909-104529.

[16] L. Hu, D. S. Hecht, and G. Grüner. "Percolation in Transparent and Conducting Carbon Nanotube Networks". In: *Nano Letters* 4.12 (2004), pp. 2513–2517. DOI: 10.1021/nl048435y.

[17] Robert Keim. *Thin and Thick Film*. Visited: 14/09/2023. 2022. URL: https://eepower.com/resistor-guide/resistor-materials/thin-and-thick-film/#.

[18] *Autodesk makes CAD design software for land surveyors, civil engineers, project managers, and construction professionals.* Visited: 14/09/2023. 2021. URL: https://www.autodesk.com/solutions/cad-design.

[19] *Learn about the benefits of using CAD software to create your next design.* Visited: 14/09/2023. 2021. URL: https://www.autodesk.com/solutions/cad-software.

[20] TrustRadius. *Best Computer-Aided Design (CAD) Software*. Visited: 14/09/2023. 2023. URL: https://www.trustradius.com/computer-aided-design-cad.

[21] Dirk P. Kroese et al. "Why the Monte Carlo method is so important today". In: *WIREs Computational Statistics* 6.6 (2014), pp. 386–392. DOI: https://doi.org/10.1002/wics.1314.

[22] Rodrigo Simões Camara LEÃO. "Distâncias em Grafos Geométricos Aleatórios e Suas Aplicações Ao Problema da Localização em Redes de Sensores". PhD thesis. 2010, p. 4.

[23] Qiguan Wang and Hiroshi Moriyama. "Carbon Nanotube-Based Thin Films: Synthesis and Properties". In: *Carbon Nanotubes*. Ed. by Siva Yellampalli. Rijeka: IntechOpen, 2011. Chap. 23. DOI: 10.5772/22021.

[24] Dhruv Sharma et al. “A network science-based k-means++ clustering method for power systems network equivalence”. In: *Computational Social Networks* 6 (2019). DOI: 10.1186/s40649-019-0064-3.

[25] Rafat Saleh et al. “Bending Setups for Reliability Investigation of Flexible Electronics”. In: *Micromachines* 12.1 (2021). ISSN: 2072-666X. DOI: 10.3390/mi12010078. URL: https://www.mdpi.com/2072-666X/12/1/78.

[26] Frida Hovsepian. *Thick Film and Thin Film Chip Resistors - Riedon Company Blog*. Visited: 14/09/2023. 2021. URL: https://riedon.com/blog/thick-film-and-thin-film-chip-resistors/.

[27] Pistekjakub. Visited: 15/09/2023. 2010. URL: https://en.m.wikipedia.org/wiki/File:Incidence_matrix_-_directed_graph.svg.

[28] Stefan Michael Blawid. *Aula 4-4 Condições de Contorno*. Visited: 15/09/2023. 2023.